University of Alberta

TOWARDS COLLABORATION IN EVIDENCE-SUPPORTED, QUESTION-DIRECTED PROGRAM COMPREHENSION

by

Benjamin Chu ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2007

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-33220-7
Our file Notre référence
ISBN: 978-0-494-33220-7

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

I think there is a world market for maybe five computers.

– Thomas J. Watson, IBM Chairman, 1943.

To my friends and family,
This would not have been possible without your support.

Abstract

Preserving explicit forms of software system documentation that are accessible by large development teams with regular developer turnover is a difficult problem. This problem can result in temporal and spatial miscommunication, an easily lost cognitive work context, and largely unmaintainable software. The research described in this thesis hypothesizes that the problem may be addressed by using a flexible question-evidence knowledge representation methodology for documenting program comprehension focusing on three areas of need: a parallel redocumentation system, immediate dissemination of knowledge amongst a team and peripheral awareness of work context. An observational study provides insight into tool requirements and a documentation methodology for a collaborative program comprehension documentation tool. A prototype named *Pollinator* is implemented and is initially evaluated against its requirements and other tools. We find that *Pollinator* shows promise in providing support to documenting program comprehension. A small-scale user study of *Pollinator* demonstrates its potential utility to software engineers.

Acknowledgements

I would like to thank the anonymous subjects of the observational and *Pollinator* user studies for their participation, Dr. Eleni Stroulia for assistance in conducting the study, my supervisor Dr. Kenny Wong for his conscientious advice and insight, the members of my examination committee, my friends for providing much needed distraction, and finally my parents and family for their support and motivating words, and without whom this thesis would never have been completed.

I would also like to thank my fellow grad students, in particular Dean Cheng, Johnny Huynh, Xin Li, Hossein Mohtasham, Dabo Sun and Daniel Moise for their advice and friendship. I also thank Anjan Sen for the use of his online voting system as the subject of the *Pollinator* user study tasks.

Table of Contents

1 Introduction . . . **1**
- 1.1 Problem . . . 2
- 1.2 Hypothesis . . . 4
- 1.3 Contributions . . . 4

2 Background . . . **6**
- 2.1 Task Strategies: Systematic and Opportunistic . . . 7
- 2.2 Top-Down and Bottom-Up Activities . . . 7
- 2.3 Integrated Meta-model . . . 8
- 2.4 Representing Program Comprehension . . . 8
- 2.5 Collaboration in Software Engineering . . . 10

3 Related Work . . . **13**
- 3.1 Cognitive Support . . . 14
- 3.2 Collaboration . . . 15

4 Observational Case Study . . . **18**
- 4.1 Experimental Design . . . 19
- 4.2 Coding Scheme . . . 20
- 4.3 Observations . . . 22
- 4.4 Analysis . . . 22
- 4.5 Study Shortcomings . . . 31

5 Prototype: Pollinator . . . **33**
- 5.1 Approach to Problem . . . 34
- 5.2 A Knowledge Representation Model . . . 34
- 5.3 Motivation for Tool . . . 35
- 5.4 Requirements . . . 39
- 5.5 Design and Implementation . . . 41

6 Preliminary Evaluation . . . **50**
- 6.1 Methodology . . . 51
- 6.2 Feature Set . . . 51
- 6.3 Tool Comparison . . . 57
- 6.4 Heuristic Evaluation of Usability . . . 58
- 6.5 Cognitive Walkthrough of Usability . . . 61
- 6.6 Conclusions . . . 72

7 Secondary Evaluation: Pollinator User Study . . . **73**
- 7.1 Objective . . . 74
- 7.2 Design . . . 74
- 7.3 Observations . . . 76
  - 7.3.1 Participant Profiles . . . 76
  - 7.3.2 General Observations . . . 77
  - 7.3.3 Study-specific Observations . . . 78
- 7.4 Analysis . . . 81

7.4.1 Task Analysis . . . 81
7.4.2 Participant Feedback . . . 83
7.5 Threats to Validity . . . 85
7.6 Study Conclusions . . . 86

8 Conclusions and Future Work 88

Bibliography 91

A Implementation Details 94
A.1 Pollinator: Extensions . . . 95

B User Study Materials 97
B.1 Solicitation Letter . . . 98
B.2 Consent Form . . . 100
B.3 User Study Handbook . . . 103
B.3.1 Content/Overview . . . 103
B.3.2 Study Protocol . . . 103
B.3.3 Restrictions . . . 106
B.3.4 History . . . 107
B.4 Pilot Tasks . . . 108
B.5 Control Tasks . . . 114
B.6 Experimental Tasks . . . 120
B.7 Pre-Study Questionnaire . . . 126
B.8 Post-Study Questionnaire (EXPERIMENTAL) . . . 127
B.9 Post-Study Questionnaire (CONTROL) . . . 129
B.10 Honorarium Acknowledgement . . . 132
B.11 Ethics Approval Certificate . . . 133

C User Study Task Analysis 134

D User Study Participant Feedback 155

List of Tables

4.1 The seven most frequent events of session 1 22
4.2 The seven most frequent events of session 2 22
4.3 The seven most frequent events of session 3 23
4.4 Most frequent event sequences (overall) 23
4.5 Event occurrences by subject (session 1) 25
4.6 Event occurrences by subject (session 2) 25
4.7 Event occurrences by subject (session 3) 26
4.8 A model of observed comprehension scenarios 29
4.9 Classification of episodes into comprehension scenarios 29
4.10 Interepisode event counts 30

6.1 Comparative summary of Pollinator and related tools 56
6.2 Open-source project Azureus at a glance 63

7.1 Participant profiles 77
7.2 Task scores by group 81
7.3 Time spent for each task 82
7.4 Participant feedback 83

List of Figures

2.1 Collaboration need hierarchy 11

4.1 Observed phases of comprehension 28

5.1 Screenshot of the *Pollinator* tool environment 41
5.2 *Pollinator* communications architecture 42
5.3 *Pollinator* project explorer 43
5.4 Awareness view of *Pollinator* environment 43
5.5 Team member detail in awareness view 43
5.6 *Pollinator* knowledge base view 44
5.7 *Pollinator* goal explorer view 44
5.8 *Pollinator* user chat view 45
5.9 *Pollinator* related files view 45
5.10 *Pollinator* editor annotation 46
5.11 *Pollinator* decorations in Eclipse 47

6.1 A portion of the many Java packages in the Azureus project 64
6.2 *Goal Explorer* and source editor after adding `mytorrents` evidence . . . 68
6.3 The final state of the comprehension tree 70

C.1 Pilot study task 1 result 135
C.2 Control study answer to Task 1 137
C.3 Experimental study task 1 result 138
C.4 Reference answer for task 1 139
C.5 Pilot study task 2 result 139
C.6 Control study answer to Task 2 141
C.7 Experimental study task 2 result 143
C.8 Reference answer for task 2 143
C.9 Pilot study task 3 result 145
C.10 Control study answer to Task 3 146
C.11 Experimental study task 3 result 147
C.12 Reference answer for task 3 148
C.13 Pilot study task 4 result 149
C.14 Control study answer to Task 4 150
C.15 Experimental study task 4 result 151
C.16 Reference answer for task 4 151
C.17 Pilot study task 5 result 153
C.18 Control study answer to Task 5 153
C.19 Experimental study task 5 result 155
C.20 Reference answer for task 5 155

Chapter 1

Introduction

1.1 Problem

Many of the software systems being used in industry today are legacy systems. By the virtue of their age, these systems also tend to be large, complex, and perhaps poorly documented. This poses problems when considering that software systems typically need constant maintenance and updating. Consider that more than 50% of all professional programmer time is spent on the task of software maintenance [27]. Here we define software maintenance as activities that involve modification or updates to previously written systems. Software maintenance is also the longest phase in the life-cycle of a software system, spanning years and sometimes lasting over two decades [17].

Furthermore, it has been stated that 40-60% of the time spent on software maintenance is dedicated to program comprehension activities [42]. Program comprehension can be described as the task of building mental models of a software system at various abstraction levels, from low-level models of the code to those of the problem domain, for the purposes of software maintenance, evolution, and re-engineering. Therefore, program comprehension forms an integral part of the software engineering process and is a worthwhile area of research.

The size of software maintenance teams is also an important issue, as many teams are made up of a large group of developers whose membership may span back in time for years or even decades. Turnover in personnel is a major problem for large and complex software projects [7].

Difficulties also arise in software maintenance and enhancement tasks on these large, complex software systems where explicit documentation is missing or experience is lacking on the part of the software engineer [22]. The importance of documenting knowledge cannot be understated in these situations as it is acknowledged that the code is only one contributor to the understanding of a program [41].

If there is no documentation process, there may be an over-reliance on the implicit knowledge that exists in the experienced minds of seasoned developers. This can be problematic as a project ages and developers move on or are separated by large distances, limiting the use of rich communications media such as face-to-face contact. In fact, previous studies have reported that the frequency of communication between engineers whose offices were separated by distances of 30 meters or more dropped to levels nearly equivalent to those of engineers separated by distances measured in miles [1], suggesting that communication is very sensitive to the degree of co-location. Large turnover in personal communications networks was also seen, even over periods as short as seven months [10]. Even if experienced developers are accessible, past design decisions and understanding may have been forgotten over time. Existing documentation also very frequently becomes stale as a system evolves, leading to bad assumptions and confusion over what is still true and relevant.

The lack of documentation and inability to communicate knowledge effectively may make a software system difficult to understand or modify by engineers who are unfamiliar with all aspects of the system. Additionally, program comprehension tools and methods that are not tightly integrated into the development process can contribute to the problems of an environment in which frequent losses of work context already occur [8]. The software ultimately becomes difficult, if not impossible to practically maintain.

The objectives of this research attempt to address three areas of need that comprise a large part of the problem by:

- providing a semi-structured program comprehension documentation methodology using a hierarchical question-evidence knowledge representation,
- allowing and promoting redocumentation of a system in parallel with a developer's normal software development workflow by implementing it in a widely-used integrated development environment [19], and
- sharing knowledge throughout a team of developers by dispersing it to other team members as soon as new understanding is developed.

This thesis begins with a brief background on program comprehension theories and ideas that have been previously studied. Next, there is discussion on related work completed in the area of cognitive support in program comprehension and research into collaboration needs in software development. Then, an observational case study is presented to illustrate an instance of how program comprehension is done by a collaborative pair of programmers with typical off-the-shelf development tools. Based on an analysis of our observations, we hypothesize an approach to documenting knowledge for collaborative program comprehension and identify a set of requirements for a cognitive support tool. A prototype tool we call *Pollinator* is presented and evaluated by comparing it against other tools and also evaluating its fulfillment of the requirements derived earlier by examining its features and characteristics. A small reflective case study is completed to demonstrate the use of Pollinator and provide initial impressions of the tool's capabilities, followed by a user study providing further evaluation of *Pollinator*'s usability and utility. The thesis concludes with a summary of the contributions and future work.

1.2 Hypothesis

We hypothesize that the problems associated with traditional program comprehension on a large system with several developers (spatial and temporal miscommunication, easily lost work context, and unmaintainable software) can be addressed or managed through preserving knowledge and work context during development tasks via explicit knowledge representation structures and automatic distributed persistence of knowledge, alongside the provision of both active and passive peripheral team awareness.

We propose that the approach to documenting program comprehension in collaborative environments we have outlined and implemented in the tool can cognitively aid developers in preserving knowledge and work context during development tasks while promoting the dissemination of knowledge through the dimensions of time and space.

1.3 Contributions

Overview of contributions:

- We present an observational case study of a pair-program comprehension task and attempt to model the methodology used by the developers in their work. This involved creating a coding scheme from the transcribed events and further generalizing the events into one of four program comprehension "phases".

- We derive a set of requirements for a tool that may be used in documenting program comprehension activities involving one or more developers based on our model of program comprehension phases and our direct observations from the case study.

- *Pollinator* is presented as an implementation of the requirements driven by the observational case study. The prototype tool integrates as a plug-in to the Eclipse[1] development environment, providing facilities for documenting program comprehension with a focus on supporting collaboration among multiple developers and long term preservation of knowledge.

- The prototype tool *Pollinator* is evaluated by a small tool usage experiment to judge its utility and usefulness towards fulfilling the hypothesis. The evaluation includes a comparison of the tool to similar projects and how they differ in their higher-level objectives and/or approaches to the problem. *Pollinator* is also judged against the derived requirements for such a tool. Heuristic and cognitive walkthrough evaluations are also performed on the design of the tool's user interface and usability.

---

[1] Eclipse - an open development platform: http://www.eclipse.org/

- *Pollinator* is evaluated through the execution of a small-scale user study in order to assess its usability and utility. In order to make comparisons to the original observational case study, the user study looks at pairs of participants working together on program comprehension documentation tasks both with *Pollinator* and in a control group without it. Observations of the participants and feedback provided by the participants allows a fair assessment of *Pollinator* as it could be used in research and industrial software engineering settings.

Chapter 2

Background

This chapter describes several previous theories and approaches to the study of program comprehension and the role of collaboration in software engineering. There are many cognitive theories on program comprehension [3] [14] [16] [38], and those discussed here are the most relevant to the work of this thesis.

2.1 Task Strategies: Systematic and Opportunistic

In the *systematic* approach to program comprehension, as observed by Littman *et al.*, programmers look at each line of code in detail, performing an extensive analysis of a program's control and data flows [16]. In this manner, the programmers were able to gain a global view of the program's behavior before initiating any modifications. The drawback to this meticulous approach is that it can be infeasible to use on large-scale systems [44]. In contrast, the *opportunistic* approach has programmers looking at different pieces of code on an as-needed basis according to the particular task at hand. This approach minimizes the amount of time spent studying the system, however the out-of-order, localized focus results in the acquisition of only static knowledge (structural information about the program) without the causal knowledge (interactions between components of the program during execution) needed to correctly perform most code modification tasks. Thus, code modifications were observed by Littman *et al.* to be more successful when a systematic approach was taken [40].

2.2 Top-Down and Bottom-Up Activities

Brooks observed that *top-down* comprehension is used when the system's domain is familiar [3]. Soloway and Erlich say this process is used when the code or type of code is familiar [38]. Koenemann and Robertson assert that program comprehension is a top-down, goal-oriented, and hypothesis-driven problem solving process [14].

In this theory, comprehension occurs as the reconstruction of mappings from the problem domain into the programming or code domain that was initially formed during the original programming process. The top-down process begins with the formation of a primary hypothesis about the global structure of the program, often created as soon as an expert programmer discovers the name or nature of the system under study. As it is usually impossible to verify this hypothesis directly against the program code, the process continues with the generation of subsidiary hypotheses forming a hierarchical structure. Lower-level hypotheses consider increasingly more concrete implementation details, thus the hierarchy bridges the gap between the application domain and programming domain knowledge. At the lowest levels in the hierarchy, hypotheses are verifiable against program code or documentation [3]. The hierarchy is usually created in a top-down, depth-first manner, so that hypotheses on the same level are resolved one at a time, reducing the overall memory load.

In *bottom-up* comprehension, programmers begin by examining individual lines of code, assigning meanings to each line encountered. These meanings are built up by aggregation into larger and larger groups, through a process known as *chunking*. This process is repeated on the code base until the entire program is understood. Brooks asserts that bottom-up comprehension is really a degenerate case of the top-down process [3].

2.3 Integrated Meta-model

Usually a mixed approach to comprehension is taken, as observed by von Mayrhauser *et al.* [44]. Top-down and bottom-up activities are complementary and have been used together in unified models, such as the integrated code comprehension meta-model [30]. In this model, there are three different comprehension processes (program, situation, and top-down model) along with a knowledge base. The systematic and opportunistic task strategies can be used to refine the tasks performed in each of the three processes.

2.4 Representing Program Comprehension

In addition to modeling the general approaches to program comprehension, there has been much research into structural models of program comprehension activity and knowledge.

Letovsky's model of program comprehension divided a programmer's understanding into three macro activities: inquiries, reading or scanning, and conversational exchanges with the interviewer [15]. The latter component could largely be disregarded as it was only applicable to the particular study methodology and there was no evidence of cognitive events in those exchanges being important to the understanding process. Reading and scanning was the assimilation of materials without any apparent difficulties, as might be expected in the execution of inquiries. Inquiries envelope a set of actions related to a particular comprehension topic: questions, conjectures, and searches. In an ideal inquiry, the developer will be browsing through code or reading some text that prompts him to ask a question. The developer then explicitly conjectures an answer to the question (a guess or hypothesis). An answer is searched for by looking through code, documentation, and other artifacts. This search may be augmented or even supplanted merely through detailed reasoning about the system under study. However, Letovsky found that this inquiry model was an idealization because frequently one or more of the inquiry components would be missing. For example, an explicit question was not always asked, as the subject may have implicitly formed the problem in their mind. Also, sometimes no conjecture was made explicitly because either the subject did not possess enough knowledge to make a conjecture or one was not uttered aloud. Sometimes guesses were accepted as conclusions, precluding the need for searches or detailed reasoning. There was also evidence for aborted inquiries, where the search or conclusion were missing or abandoned for whatever reason. Letovsky also developed a taxonomy to describe the types of questions programmers would ask during the comprehension process. The taxonomy was made up of why, how, what, whether, and discrepancy (over perceived inconsistency) questions.

Pennington researched the comparative efficacies and dominances of mental programming knowledge representations [27]. The two representations under comparison were procedural (control-flow) and functional (goal hierarchy). Her research was based on approaching computer programs as text and applying already established text comprehension theories to the analysis. She suggests two kinds of programming knowledge: text structure and plan knowledge. In general, Pennington found that the complete comprehension of software systems required the understanding of multiple relations between parts of the "text" that are difficult to view simultaneously. The results of the research suggested that the procedural knowledge representation rather than functional formed the initial basis of expert programmers' mental representations. However, additional results suggested that the programmer's task goals were found to influence the relations dominating mental representations later in the comprehension process.

In Pennington's research, computer program stimulus structures represent abstractions of the text (program) and are intended to illustrate features of the text, but are not mental entities. These structures may or may not be detected during comprehension. Control flow stimulus structures are sequences of statements and certain keywords that provide information about a sequence of statement execution. Data flow stimulus are changes or constancies in meaning or value associated with names of program objects throughout the course of the program code.

Two abstractions of the program text of particular interest are the goal hierarchy and data flow representations. The goal hierarchy represents the goals of the program, abstracting functions but providing little explicit information on how the goals will actually be accomplished (i.e. no implementation details). Data flow knowledge representations are shown in terms of the program processes acting as transformative agents on the initial data objects and turning them into the outputs of the program. It is possible for the goal hierarchy to be recovered from the data flow structure with knowledge to infer the subgoals. The data flow structure shows more than the goal hierarchy on the interactions between data objects and ordering of operations.

2.5 Collaboration in Software Engineering

The collaborative nature of software engineering means that program comprehension is necessarily a collaborative activity as well. Here we describe some background of the role of collaboration in software maintenance and some of the research that has been done in regards to program comprehension as a collaborative activity.

Software engineering has been cited as a profession where typical system developers spend 70% of their time working with others, and team activities comprise 85% of the cost of development in large software systems [43]. Holt takes the position that software architecture itself is a mental model shared among people responsible for the system. In other words, the key purpose of software architecture is to facilitate team communication and understanding [11]. Collaboration bleeds into other, specialized areas of software engineering, such as software maintenance, which has been described as a highly collaborative activity requiring coordination of the work of current and previous maintainers through direct communication, documentation, and the source code itself [17]. Software maintenance programming also tends to be a high-turnover position, whether it is developers seeing it as a stepping stone to better things or management using it as a training ground for new hires [20], so in practice preservation of knowledge needs to survive through several generations of team members. Lougher concludes that support must be provided for software maintenance as a group activity.

The theory of distributed cognition (DC) [45] applied to software engineering tools implies that program comprehension, among other software engineering activities, is a process that involves spreading cognition among multiple humans and artifacts, analogous to the way computations are divided among separate discrete processing units in distributed computational systems. Cognition-related data, such as goals, plans, and ideas, may be offloaded into an external memory, out of the developer's mind. This external memory may be distributed among a system's artifacts and requires a method or representation to reconstruct the original cognitive processes.

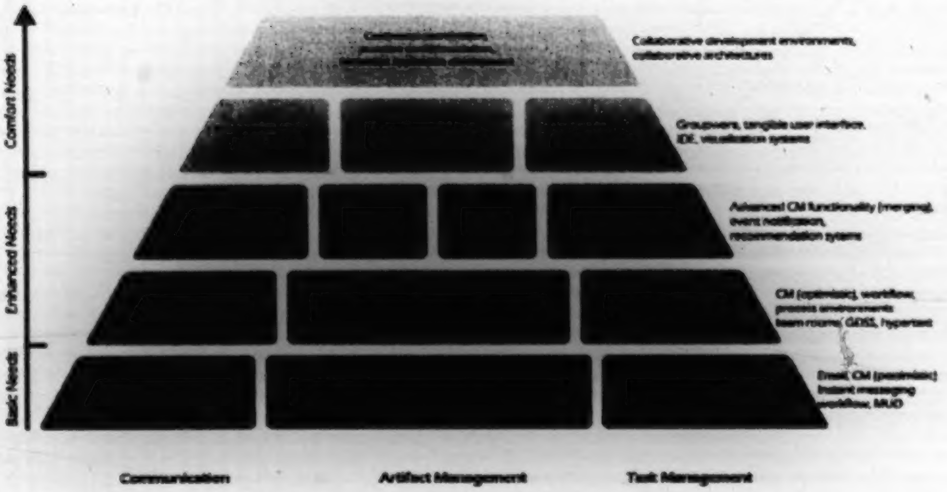

Comfort Needs
Enhanced Needs
Basic Needs
Collaborative development environments, collaborative architectures
Groupware, tangible user interface, IDE, visualization systems
Advanced CM functionality (merging), event notification, recommendation systems
CM (optimistic), workflow, process environments team rooms, GDSS, hypertext
Email, CM (pessimistic) Instant messaging workflow, MUD
Communication
Artifact Management
Task Management

Figure 2.1: Collaboration need hierarchy

Pair programming can be described as a form of distributed cognition. System knowledge is shared between the members of a team as partners are frequently changed, making it rare that both partners are new to the code being studied and/or modified [46]. Pair programmers can monitor and learn from each other's comprehension strategies, as the practice of thinking aloud in these types of collaborative situations makes the strategy explicit to the listening partner. Therefore, pair programming not only shares understanding, but shares plans of attack on the problem [42].

van Deursen posits several directions for research on program comprehension in pair programming. Two of particular interest are the question of how working in pairs affects program understanding and how the discussions between pair partners can be utilized to improve our models of how program comprehension works in the developer's mind.

In laying out a need-based collaboration tool classification framework, Sarma adapts Maslow's hierarchy of needs [21] to delineate collaboration needs that are broadly classified into those of basic, enhanced, and comfort [37] (see Figure 2.1). Within the three broad categories, there are currently five layers describing attributes of tools that provide for those needs. The area of comfort needs is where there is still much work to be done, according to Sarma, and it is comprised of the two highest-level layers. Layer four includes tools that provide passive awareness of parallel activities, maintenance of dependencies between artifacts and people, and contextual awareness. At the fifth layer, tools provide continuthat provide passive awareness of parallel activities, maintenance of dependencies between artifacts and people, and contextual awareness. At the fifth layer, tools provide continuous coordination, proactive conflict resolution, and "frictionless" application development. Sarma argues that many tools become "stuck" in the bottom layers of the hierarchy because they tend to rely on a lot of functionality that is created from scratch rather than building on existing infrastructure.

Chapter 3

Related Work

3.1 Cognitive Support

There are several program comprehension tools that provide cognitive support to users within an integrated development environment such as Eclipse.

FEAT [32] and ConcernMapper [33] allow a user to manage mappings of scattered concerns to source code elements. The concern graph, a structure representing groups of software concerns, is argued as being a more effective software representation than lines of code because it abstracts away implementation details. However, there is no provision for the nesting of concerns hierarchically as suggested for goals and questions in this thesis.

JASPER (Java Aid with Sets of Pertinent Elements for Recognition) takes a similar approach to managing functional groupings in source code, but also considers other artifacts [5]. Again, the basic idea is that individual lines of code by themselves mean little since functionality is typically implemented and documented throughout many different source modules and pieces of documentation. In JASPER, artifacts representing task-relevant data can be grouped together into *working sets* that represent a particular task, such as "Add thickness slider", for easy reference. JASPER is implemented as an Eclipse plug-in. Artifacts are visible in a single view within its particular context. The goals are to reduce time spent on redundant navigations, while facilitating multitasking, interruption management, and sharing task information with other developers.

Hipikat [41] supports the formation of implicit group memory from the information in a project's archives (CVS, email, bugzilla, etc.) and recommends artifacts from those archives based on a programmer's work context or explicit search terms. While the idea of a group memory is emphasized in Hipikat, as a tool it does not address the issue of real-time collaborative awareness and knowledge dissemination as elaborated in the requirements for the *Pollinator* comprehension support tool.

NavTracks [12] aids code browsing by keeping track of a programmer's navigation history and forms associations between files so that files related to the current context may be recommended as relevant to the user. This type of cognitive support is focused on building a comprehension model at the level of the filesystem, whereas our tool is more fine-grained and allows a programmer to build a model using individual source code-level elements.

TagSEA [39] is an Eclipse plug-in that allows a programmer to create waypoints (analogous to waypoints in geographical navigation), that mark locations of software model elements through the tagging of elements inline with the source code. These waypoints can then be navigated by using the previously generated tags in an integrated waypoint viewer.

This approach focuses on the idea of "social tagging" that is used in many online web applications for the purposes of assigning objects to multiple user-defined categories. Thus each individual object may be viewed from several different perspectives instead of being restricted to occupying a single space in a particular taxonomy. This allows for a large degree of flexibility in choosing how to identify the points of interest in a software system. *Pollinator* uses a slightly more structured hierarchical approach in the construction of a mental model, but aims to make the discovery of knowledge as accessible and non-duplicative as possible.

Mylar is a tool that reduces the cognitive load of the current work context by putting a task filter on the types of information presented to the user in the Eclipse workspace environment [13]. It analyzes the most frequently used artifacts of the workspace and other work activity to evaluate the relevancy of the information contained therein. The effort required to complete software maintenance tasks or projects is reduced and reused in this manner by making the task context explicit.

Parnin *et al.* describe research done in assessing the value of using data obtained from recorded interaction histories (contexts) as a way of improving the ability of a programmer to recover the mental state associated with tasks [26]. A recommendation system using some chosen prefetch algorithm could allow the programmer to recover previous work contexts. The evaluation of data from case studies showed that discarding the *least recently used* method from a context was the best in terms of recovery performance, most likely due to the dominance of temporal locality in deciding method relevance.

The evaluation of these cognitive support tools is important, and Walenstein describes a theory-directed approach to measuring the cognitive support provided by program comprehension tools [45].

3.2 Collaboration

While the work described in the previous section is targeted towards cognitive support of software engineering in the context of program comprehension, the work described here is specifically focused on the aspect of *collaboration* in software engineering.

Booch and Brown provide a rationale for collaborative development environments (CDE), which they describe as virtual spaces where all project stakeholders may participate to carry out some task such as creating an executable deliverable and its supporting artifacts [2]. The team-centric nature of CDEs set them apart from traditional developer-centric IDEs which were only designed with supplemental support for collaborative features.

Sarma describes several collaboration tool projects that address the fourth layer of collaboration needs that were discussed in Chapter 2. Sangam is an Eclipse plug-in described as a collaborative tool for facilitating distributed pair programming over the Internet using features like screen and application sharing [35]. The Eclipse Communication Framework (ECF) is an architecture for supporting the development of distributed Eclipse-based tools and applications [29]. JAZZ is a collaborative application development environment that is meant to extend the Eclipse Java environment by providing for contextual collaboration [4]. The JAZZ environment provides elements of collaboration such as user awareness, discussion boards, instant messaging, linkage of chat and messages to code, logs of team events, and more. Palantír is an Eclipse plug-in tool for supporting workspace awareness by allowing developers to monitor parallel workspace activity without having to switch mental context [31]. It supports a collaborative workstyle by showing those artifacts which have been changed, alongside who changed them and the degree to which they were changed. The focus of Palantír is the avoidance or early detection of conflicts during parallel activities, reducing the effort needed to address problems later.

Ma's research into Adoption Centric Reverse Engineering (ACRE) places particular emphasis on collaboration in reverse engineering within existing, well-used software environments, such as Lotus Notes [18]. Ma's thesis lists several requirements for a tool to support team collaboration during software maintenance tasks that were originally developed as evaluable aspects of cooperative functionality in CASE (Computer Assisted Software Engineering) tools by Henderson [9].

1. Maintain a dialog with team members
2. Allow the team to simultaneously work on a single task
3. Send messages/e-mail to team members
4. Concurrent use of dictionary and diagrams
5. Group interaction support (e.g. brainstorming)
6. Attach electronic notes to objects
7. Anonymous feedback or input
8. Notify engineers if a design change affects their work

9. Build a catalog of macros accessible by the team

Lougher's position that software maintenance must be supported as a collaborative activity was motivated by the idea that the extended time period over which maintenance occurs makes direct communication between team members difficult or impossible [17]. This led his research into a system that supports long-term collaboration in maintenance by allowing the maintenance rationale to be captured and shared by engineers through a large range of unconstrained documentation facilities and techniques such as hypertext to attach comments to relevant source components. Lougher dubs this system a type of "annotative collaboration".

There has also been research into classification frameworks for collaborative tools, used to place the tools in context as well as serving as a guide to developers on making the best choice for a tool for their purposes [36]. Each framework may have a different classification focus, such as providing a taxonomy to compare tools, or comparisons along functionality, collaboration approach, or user effort.

Cook's survey of collaborative software engineering notes that the area is comprised of many areas of research, including: distributed systems, configuration management, human computer interaction, groupware systems, and software visualization [6].

Chapter 4

Observational Case Study

In order to gain insight into the needs of the documenting process in collaborative program comprehension, an observational case study was undertaken.

4.1 Experimental Design

During the study, two graduate-level computing science subjects were observed as they worked together as a pair on understanding the Perl interpreter. Specifically, their goal, which was independent of our study, was to find where the Perl interpreter could be instrumented to gain access to the abstract syntax tree of a parsed Perl program. The observational nature of the study is based on the principle that pair programming makes it possible to overhear other programmers' code comprehension discussions and join them when needed [42], or in the case of one pair of programmers, the discussions and activities occurring between the two subjects may explicitly represent their comprehension activities, negating the need for a think-aloud protocol or prompting / interviewing by the experimenter.

The tools used for comprehension were those chosen by the subjects, and included the use of Microsoft Visual Studio 6.0 for code browsing and compilation, the Windows notepad program for writing small test cases in Perl, and paper notebooks to write down thoughts and knowledge gained from each comprehension session.

The sessions were recorded using a digital video camera mounted on a wheeled tripod situated behind and just to the side of the subjects' usual workstation. The view of the camera alternated between the subjects and detailed shots of the screen contents. Most non-computer related actions performed by the subjects were captured by the camera and all sound was also recorded. The observer maintained total silence throughout the sessions and did not interfere with the participants' actions.

In total, we observed four sessions and decided that only the first three sessions were focused on program comprehension, so the fourth was excluded in the final analysis. The sessions were 1:18, 1:53, and 2:43 (hours:minutes) in length, respectively. The total video used for the analysis was five hours and 54 minutes in length.

The sessions were transcribed into discrete narrative segments of events corresponding to noteworthy activity such as an explicit question asked or a period of code browsing. The temporal resolution of the events recorded were at the *meso-scale*, a measure of magnitude that groups the detail of the sessions into forms that are useful for comprehensive analysis without focusing on irrelevant observations [15].

Events irrelevant to the program comprehension task, such as pleasantries exchanged at the beginning of each session, were eliminated from the final compilation of event occurrences. This ensured that only events pertaining to the activities of program comprehension were under analysis.

As a major focus of the study was on analyzing the collaborative aspects of comprehension, each event was tagged with the identification of the subject initiating the event. Since the subjects shared a single computer, the possession of computer control was also recorded as it switched back and forth between subjects.

Events occurring simultaneously (such as speaking and reading) were recorded as separate discrete events, but grouped together into "episodes" or phases of activity. These episodes were classified into one of four high-level comprehension scenarios (orientation, goal creation, question asking and answering, and goal resolution) as described by our abstracted model in Table 4.8, which was derived from our observations in the study.

4.2 Coding Scheme

The development of a coding scheme was necessary in order to classify the discrete events of each session. By analyzing the taped footage, we compiled a list of the most common events observed in the sessions and developed a classification schema, described here. The events are partitioned into three main top-level macro activities of: *reading*, *writing*, and *speaking*.

Reading Codes

- `CMP` indicates when the reader is looking at something on the computer that is not covered by the other reading codes. e.g. *Subject is using Windows Explorer to find files related to the task.*
- `CMP/CB` indicates when a reader is browsing through code on the computer. e.g. *Subject is looking for the macro definition of SV.*
- `CMP/Doc` indicates when a reader is looking at documentation on the computer. e.g. *Subject is reading the online documentation for the Perl interpreter.*
- `Doc` indicates when a reader is looking at hardcopy documentation or notes. e.g. *Subject is looking at his notes from the previous session.*

Writing Codes

- `CMP/Code` indicates when the writer is programming code using the computer. e.g. *Subject is writing a Perl script to observe the interpreter state.*

- `Notes` indicates when the writer is writing notes in their notebook. e.g. *Subject is drawing a diagram showing the relationship between the SV data structure nodes.*

- `Notes/Demo` indicates when the writer is writing notes in their notebook to illustrate something to another person. e.g. *Subject is writing an example of the Perl hash variable type.*

Speaking Codes

- `DIR` indicates when the speaker is issuing a directive or imperative in order to get a listener to take some course of action. e.g. *Subject tells other subject to do a file search for parse() function.*

- `EXP` indicates when the speaker is explaining something to a listener, illustrating a piece of evidence or imparting previously learned knowledge, but is not responding to an explicitly asked question. e.g. *The structure of the Perl interpreter depends on the dynaloader and miniperl modules.*

- `FF` indicates when the speaker is participating in discussion that is largely unstructured or is saying something in a way that is not covered by the other speaking codes. e.g. *Subjects are reorienting themselves to get at a common point from which to start.*

- `HYP` indicates when the speaker is proposing or generating a hypothesis, either implicitly or explicitly. e.g. *The subject says: "I think the function clears the stack before calling parse_func()."*

- `QA-Q` indicates when the speaker is asking a question of a listener. e.g. *"How are Perl variables represented in memory?"*

- `QA-A` indicates when the speaker is answering an explicit question asked by a listener. e.g. *"Perl variables are held in a stash data type."*

- `ST` indicates when the speaker seems to be talking to themselves or thinking aloud without directing their words to another person. This may occur when the speaker is paraphrasing a concept to reinforce its understanding. e.g. *"So the stash data type holds variables, but the SV field of the data structure is not used."*

Event Description	**Event code**	**Frequency**
Code Browsing	*Reading:* `CMP/CB`	74 (30%)
Free Form Discussion	*Speaking:* `FF`	59 (24%)
Explanation	*Speaking:* `EXP`	24 (10%)
Online Documentation	*Reading:* `CMP/Doc`	17 (7%)
Question Asked	*Speaking:* `QA-Q`	17 (7%)
Paper Documentation	*Reading:* `Doc`	14 (6%)
Question Answered	*Speaking:* `QA-A`	13 (5%)

Table 4.1: The seven most frequent events of session 1

Event Description	**Event code**	**Frequency**
Code Browsing	*Reading:* `CMP/CB`	79 (29%)
Free Form Discussion	*Speaking:* `FF`	54 (20%)
Explanation	*Speaking:* `EXP`	34 (12%)
Online Documentation	*Reading:* `CMP/Doc`	25 (9%)
Question Asked	*Speaking:* `QA-Q`	23 (8%)
Question Answered	*Speaking:* `QA-A`	21 (8%)
Notes	*Reading:* `Notes`	11 (4%)

Table 4.2: The seven most frequent events of session 2

4.3 Observations

During the course of the sessions it became evident that one of the subjects had previously partly studied the code of the Perl interpreter and was therefore regarded as a technical expert. The other subject frequently consulted the technical expert on the details of Perl interpretation while also serving as the director of the higher-level comprehension activities [46]. The activities of this technical director included maintaining and directing the work context.

The collaborative nature of the observed program comprehension session meant that it was relatively easy to observe the cognitive activities of the participants without the need for the observer to explicitly prompt or question the participants, as communication between them was frequent and mostly explicit.

4.4 Analysis

Session Synopses.

Here we present a brief summary of each program comprehension session used for analysis.

Session 1.

Event Description	**Event code**	**Frequency**
Free Form Discussion	*Speaking:* `FF`	115 (31%)
Computer	*Reading:* `CMP`	79 (22%)
Code Writing	*Writing:* `CMP/Code`	26 (7%)
Paper Documentation	*Reading:* `Doc`	26 (7%)
Code Browsing	*Reading:* `CMP/CB`	25 (7%)
Question Asked	*Speaking:* `QA-Q`	22 (6%)
Question Answered	*Speaking:* `QA-A`	20 (5%)
Explanation	*Speaking:* `EXP`	18 (5%)

Table 4.3: The seven most frequent events of session 3

Event Code	**Following Event Code**	**Frequency (%)**
Reading: `CMP/CB`	*Speaking:* `FF`	7.7
Speaking: `FF`	*Reading:* `CMP/CB`	7.6
Reading: `CMP`	*Speaking:* `FF`	5.0
Speaking: `FF`	*Reading:* `CMP`	4.7
Speaking: `QA-Q`	*Reading:* `CMP/CB`	2.1
Reading: `CMP/CB`	*Speaking:* `EXP`	2.0

Table 4.4: Most frequent event sequences (overall)

The first observed session can be described as a "fact-finding" project, to explicitly discuss the high-level goals and orient each other on the knowledge brought to the session by each subject. Also observed, was the establishment of the subjects' representative roles as technical expert and director. As part of the initial comprehension activities, the overall architecture of the software system was discussed, and only later were more focused code browsing activities performed in detail on certain components of the code base. This was also the session where the subjects familiarized themselves with the conventions of the code and filesystem structure.

Session 2.

At this point the subjects have already become comfortable enough with the control - flow of the system in the first session to focus on detailed examinations of the data structures of interest. In particular, they are interested in where Perl variables are represented and how this relates to the overall architecture of the interpreter. So largely, the session consists of searches and examinations of data structures, finding their references, declarations, and definitions as well as the relations between them. Also looked at were the macros and functions used to access and manipulate the data structures in anticipation of extracting information from them for the task. Online documentation is referred to, as necessary, to gain insight into the architecture of the system with regards to data structure meanings and variable representation.

Session 3.

Between the second and third sessions, a prototype Perl extractor was implemented inside of the existing Perl interpreter. This session starts off with some discussion on the state of the prototype and its capabilities. Since the focus of the session is on the prototype extractor, much of the activity is spent on modifying, debugging, and understanding the extractor code. In turn, the comprehension of the Perl interpreter continues since the extractor is deeply tied to its innards. There is a back-and-forth of modifications to the code and analyzing the debug state of the extractor. The modifications and analysis of the extractor are highly dependent on the subjects' understanding of the Perl interpreter, in particular the representation of data types and macro definitions.

Events. Here we analyze the events observed in the sessions based on their frequency of occurrence. The focus of recorded events was on communication, discourse, and collaboration, not on what menus or windows were used in the tools. Tables 4.1-4.3 show the ranking of the top seven observed events in each of the three sessions used from the study. Table 4.4 shows the most frequent event types that occurred together within episodes, one after the other. These numbers were compiled by counting the frequency with which an event occurred before another event within an episode. The frequencies of event occurrences shown in Tables 4.1-4.3 and Table 4.4 are values aggregated over the two observed programmers.

In a program comprehension task, it is not surprising that the most frequent activity performed is that of looking through the code of the system being examined. Since the task was performed as a pair of programmers, it is also evident that the magnitude of free form discussion observed will be large. This is a confirmation that communication during a collaborative comprehension task is a priority of the programmers.

More interesting are the structured forms of speech observed. The explanation, questions asked, and questions answered events are all in the top seven observed events. It is these types of discourse which contribute most to learning and the acquisition of knowledge, as noted by Letovsky in his concept of the *inquiry* model [15]. The magnitude of the question and answer frequencies is significant enough for those activities to be included among the top seven events (out of the 14 event types that were observed in session 1, for example). The similar values in frequency for questions asked and questions answered suggests that there are not very many questions left unanswered.

Roles. We wanted to see if there was any difference in the types of activity performed

Macro Activity	**Code**	**S1**	**S2**
Speaking	`FF`	27 (46%)	32 (**54%**)
	`EXP`	4 (**83%**)	4 (17%)
	`HYP`	1 (17%)	5 (**83%**)
	`ST`	1 (25%)	3 (**75%**)
	`DIR`	4 (44%)	5 (**56%**)
	`QA-Q`	1 (6%)	16 (**94%**)
	`QA-A`	13 (**100%**)	0 (0%)
Reading	`CMP`	4 (**57%**)	3 (43%)
	`CMP/CB`	35 (47%)	39 (**53%**)
	`CMP/Doc`	12 (**71%**)	5 (29%)
	`Doc`	6 (43%)	8 (**57%**)
Writing	`Notes`	1 (50%)	1 (50%)
	`Demo`	0 (0%)	2 (**100%**)
	`CMP/Code`	0 (0%)	0 (0%)

Table 4.5: Event occurrences by subject (session 1)

Macro Activity	**Code**	**S1**	**S2**
Speaking	`FF`	23 (43%)	31 (**57%**)
	`EXP`	26 (**76%**)	8 (24%)
	`HYP`	3 (38%)	8 (**63%**)
	`ST`	0 (0%)	0 (0%)
	`DIR`	4 (**100%**)	0 (0%)
	`QA-Q`	1 (4%)	22 (**96%**)
	`QA-A`	20 (**95%**)	1 (5%)
Reading	`CMP`	0 (0%)	0 (0%)
	`CMP/CB`	37 (47%)	42 (**53%**)
	`CMP/Doc`	13 (**52%**)	12 (48%)
	`Doc`	0 (0%)	0 (0%)
Writing	`Notes`	1 (17%)	5 (**83%**)
	`Demo`	1 (20%)	4 (**80%**)
	`CMP/Code`	0 (0%)	0 (0%)

Table 4.6: Event occurrences by subject (session 2)

Macro Activity	**Code**	**S1**	**S2**
Speaking	`FF`	59 (**51%**)	56 (49%)
	`EXP`	17 (**94%**)	1 (6%)
	`HYP`	2 (**100%**)	0 (63%)
	`ST`	2 (**67%**)	1 (33%)
	`DIR`	0 (0%)	11 (**100%**)
	`QA-Q`	0 (0%)	22 (**100%**)
	`QA-A`	20 (**100%**)	0 (0%)
Reading	`CMP`	39 (49%)	40 (**51%**)
	`CMP/CB`	13 (**52%**)	12 (48%)
	`CMP/Doc`	0 (0%)	0 (0%)
	`Doc`	12 (46%)	14 (**54%**)
Writing	`Notes`	3 (60%)	2 (**40%**)
	`Demo`	3 (**100%**)	0 (0%)
	`CMP/Code`	25 (**96%**)	1 (4%)

Table 4.7: Event occurrences by subject (session 3)

by each subject. Therefore, the event occurrence data divided by subject is shown in Tables 4.5 - 4.7. The role of each subject did seem to determine the activity performed, to a certain degree. For example in session 1, the technical expert was responsible for 83% of the explanation events and 100% of the question answering events. This is in contrast to the technical director who performed 83% of hypothesizing, 56% of directing, and 94% of the question asking events.

Notes. We were also interested in how knowledge becomes persistent during a program comprehension session, as a cognitive support tool could greatly assist in this aspect of comprehension. The written notes of the study participants were analyzed to see if there were any major differences between the structure and style of how each participant represented their gleaned knowledge. In general, the participant serving as the technical expert wrote notes that were much more explicitly structured than those of the director. The structured note-taker made frequent use of section headings, lists, diagrams, term definitions, source code references, and procedural recipes. This contrasts with the notes of the other participant, who mostly wrote short text fragments and questions with the occasional diagram or code snippet. As a third-party observer, the structured notes seemed much easier to understand, but this does not necessarily indicate that those notes made for better persistence of comprehension. The unstructured notes may be in a form more efficiently understood by the director.

Questions. Also analyzed were the most frequently stated goals or questions asked during the sessions. Most questions asked revolved around the desire to understand the overall system architecture, the meaning of a particular function, or the location of some particular functionality or data structure.

For example, there were numerous questions phrased generally as "What does the *MACRO NAME* macro do?" and "How is *STRUCTURE NAME* represented?" and "Where is *VARIABLE NAME* modified/accessed?".

The file search facility in Visual Studio was observed to be frequently used during code browsing events. The searches were mostly for variable references and macro definitions. According to the subjects, the heavy use of macros in the Perl interpreter source code necessitated much of the frequent searching. No meaningful search terms were used to guess at where something might be located; instead, the names of files were used as indicators of where functionality was located, usually by the technical expert of the two subjects. The use of domain-specific search terms would likely have been fruitless, as the "words" used in variable and function names were usually not obviously associated with a particular meaning nor were they fully spelled out.

Phases of Comprehension. After analyzing the taped sessions, we abstracted four fundamental program comprehension scenarios that are described in Table 4.8. We classified each episode of the observed sessions into one of these scenarios or phases of activity.

The classification of episodes allowed a broader picture of the observed comprehension to emerge in a model we believe to capture the major component and action sequences of the observed program comprehension. A common sequence of scenario activity was seen after the classification was applied to the analysis of the sessions (see Figure 4.1). Each session began with a period of *orientation* in which the programmers tried to reconstruct their comprehension work context, including a restatement of current goals, knowledge, and relevant pieces of evidence. Once the comprehension context had been reestablished, a cycle of scenarios ensued, beginning with the *generation of a goal* and zero or more sub-goals. This was followed by *question asking*, evident by an explicitly interrogative statement or an action taken suggesting an implicit question. Questions were *answered* by the process of knowledge acquisition which involved searching through code and documentation artifacts, discussions and explanations between the subjects, as well as some derivation of the answer based on logical reasoning by one or both of the subjects. *Goal resolution* then took place, as information from the previous stage was analyzed in order to test a hypothesis and judge whether or not a goal had been achieved, failed, or whether further investigation (in the form of sub-goal generation) was needed. At any point during this process, it was observed that the work context of the programmers could be interrupted by an externality

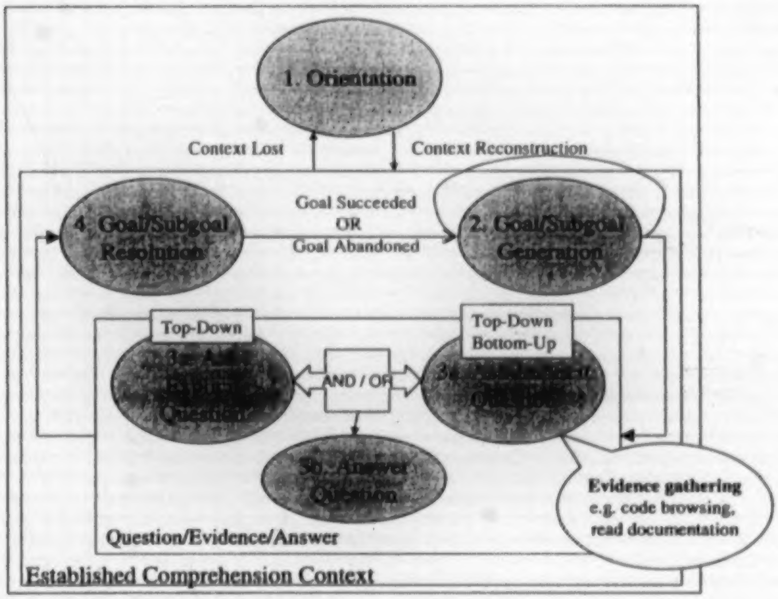

1. Orientation
Context Lost
Context Reconstruction
4. Goal/Subgoal Resolution
Goal Succeeded
OR
Goal Abandoned
2. Goal/Subgoal Generation
Top-Down
Top-Down
Bottom-Up
Question
AND / OR
3b. Answer Question
Evidence gathering
e.g. code browsing,
read documentation
Question/Evidence/Answer
Established Comprehension Context

Figure 4.1: Observed phases of comprehension

(such as a phone ringing or instant message appearing) or a sudden general confusion as to the current state of the cognitive comprehension model. The latter seemed to happen in circumstances where the programmers were deeply investigating a goal, had created a number of sub-goals, and then forgotten what their primary goal was to begin with. In this case, re-orientation took place in order to restate the current goals and rebuild the comprehension context.

Table 4.9 shows the frequency of the scenario events across the three analyzed sessions. Session 3 involved much more programming than did the other two sessions, and in particular relatively more goals are resolved in this session than the others. This may be indicative of programming goals being more easily resolved or that coding in tandem with program comprehension leads to greater comprehension success.

Scenario	Description
Orientation	Activities related to the synchronization of mindset within the group of reverse engineers. Occurs most often at the beginning of sessions, but is also observed when a loss of context happens at any other point. Usually involves a restatement of current goals and previously gleaned knowledge.
Goal Generation	Activities related to the formation of goals for a particular session or goals needed to complete an overarching goal that is already formed. These are often not explicitly stated but can be gleaned from a sequence of related events and are frequently associated with questions being asked or a decision to do something.
Questioning: Ask and Answer	Activities related to the asking of questions, either explicitly (as in the case of top-down program comprehension) or implicitly (bottom-up program comprehension). The type of program comprehension may alternate during the course of a session. Implicit questions occur during the course of gathering evidence to address a goal or explicit question. The gathering of evidence usually occurs through the *reading* macro-activity, and is often associated with searching through documentation, code, or other textual information. However, it may also occur by conversing with another individual or by synthesizing evidence from the questioner himself. The implicit question itself is as to the relevance of the evidence gathered and can also be thought of as a hypothesis. An answer to a question comes from evidence gathered and leads to the resolution of the question state.
Goal Resolution	Activities related to the resolution of a created goal by determining if that goal has been achieved through the resolution of questions asked or actions taken towards the completion of the goal.

Table 4.8: A model of observed comprehension scenarios

Scenario Type	Session 1	Session 2	Session 3	Total
Orientation	1 (1%)	1 (1%)	6 (5%)	8 (3%)
Goal Creation	5 (6%)	29 (28%)	25 (19%)	59 (18%)
Question Ask & Answer	75 (87%)	59 (57%)	75 (58%)	209 (65%)
Goal Resolution	5 (6%)	15 (14%)	24 (18%)	44 (14%)
Total	86	104	130	320

Table 4.9: Classification of episodes into comprehension scenarios

Macro Activity		Speaking							Reading				Writing		
	Code	FF	EXP	HYP	ST	DIR	QA-Q	QA-A	CMP	CMP/Code	CMP/Doc	Doc	Notes	Demo	CMP/Code
Speaking	FF	45	5	5	0	4	13	7	43	70	14	14	5	2	9
	EXP	10	8	1	0	4	2	8	7	14	7	5	1	1	3
	HYP	1	2	1	0	0	1	0	0	10	2	0	0	0	0
	ST	1	1	0	0	0	0	0	0	3	1	1	0	0	0
	DIR	3	0	0	0	1	0	1	6	11	1	0	0	0	0
	QA-Q	6	1	0	0	0	7	0	10	19	5	3	0	3	0
	QA-A	2	13	0	0	0	1	7	7	15	3	3	1	3	2
Reading	CMP	46	11	1	1	3	8	8	9	0	0	1	0	0	11
	CMP/Code	71	18	6	4	7	16	8	1	52	3	4	4	0	2
	CMP/Doc	15	12	2	0	1	1	3	0	1	3	0	1	0	0
	Doc	18	3	0	0	4	3	3	2	1	0	1	1	0	3
Writing	Notes	4	1	0	1	0	2	1	0	3	0	2	0	0	0
	Demo	2	1	1	0	0	3	3	0	0	0	0	0	0	0
	CMP/Code	10	3	0	1	0	0	3	8	2	0	2	0	0	0

Table 4.10: Interepisode event counts. Each tally in a cell indicates occurrences of the event type in the left "Code" column happening immediately in a sequence before the event type in the top "Code" row within an episode of activity.

Table 4.10 shows the counts of event sequences, events that occurred one after another over the three analyzed sessions. The number located in each cell represents the occurrences of the corresponding event type on the left column happening before the corresponding event type on the top row within an episode of activity. It is apparent from the data that free-form discussion (`FF`) and code browsing (`CMP/Code`) were deeply intertwined in this particular study. This may be expected in an environment were the two subjects are sitting next to each other and sharing a single computer. Also of interest is the observation that the majority of note-taking occurs just after free-form discussion and code browsing. This may be explained by the notion that knowledge persistence occurs only after the piece of knowledge in question has been discussed or directly observed by the subjects.

4.5 Study Shortcomings

There are two significant threats to the validity of the observational study. The small size of the participant pool is an internal threat, while an external threat is that the participants themselves may not be representative of software developers in an industrial setting.

Unlike studies where there was verbatim transcription [15] or structured interviews as conducted in [16], the analysis presented here was not based on using precise keywords like `why` or `because` to identify events like questions and answers, nor were explicit interview responses available to clearly reconcile events with their cognitive meanings.

While we believe the coding scheme and transcription of events into narrated episodes to be sufficient for illustrating several significant qualities of pair program comprehension, the analysis would certainly be enriched by the conduct of exit interviews with the study participants to clarify the meanings of some of the more ambiguous cognitive events. Also, a different coding scheme that accounts for combined events in higher detail or one that is more fine-grained and detailed may result in further insight. There is also a question of whether or not the coding scheme could be used by other analysts to reproduce the same sequence of events from the raw session data. Additionally, given the same observational data and scheme, it is uncertain whether or not a similar coding would be constructed by others. This is the general difficulty of interpreting observed events and assigning them to a specific code or category. We believe our coding scheme has been constructed so as to accommodate most event types with minimal overlap, but there are bound to be some undesirable or controversial categorizations.

Walenstein notes that tool developers will not be well-served by the theory-building activity embarked upon in this study such as deciding on the significance of various events, devising a coding scheme, and constructing models to explain the results [45]. Instead the researcher should import and use theories to alleviate the burden of using existing empirical techniques. A lot of time can be saved from coding observations in-situ by having a predetermined coding scheme.

An alternative method for conducting such a study is presented by Robillard in research on measuring cognitive activities in software engineering. The general process is to form a formal hierarchical coding scheme, and merge psychological and statistical analysis to produce an explicit cognitive model [34].

Chapter 5

Prototype: Pollinator

5.1 Approach to Problem

Our analysis of the results from the observational case study highlights some areas of deficiency in the program comprehension process used by the subjects that may be addressed through the presence of a tool that cognitively supports the documentation of comprehension. By integrating the knowledge preservation mechanisms into the development environment and sharing comprehension among members of a project we may be able to reduce the problems of temporal and spatial miscommunication. In this manner, the work context can be made more explicit and easily maintainable throughout a particular task.

Therefore, in this chapter of the thesis we use the observations gleaned from the study to motivate a set of requirements for a cognitive support tool in collaborative environments. The requirements are then used to develop such a tool, whose design and implementation is described. The chapter concludes with an evaluation of the tool.

5.2 A Knowledge Representation Model

As described earlier in the background chapter, there are several different ways to model or represent program comprehension knowledge. Based on our observations in the study we present a knowledge representation model that is hypothesized could cognitively aid the preservation of program comprehension knowledge in collaborative development environments. We believe the model to closely follow the structures and program comprehension activities as observed in the initial case study.

Cognitive models of program comprehension may define a structure to aid in the construction of a mental model, comprised of elements and strategies. We define the terminology of the elements and describe the overall comprehension structure derived partly from the literature and our own observations, which are described in the next section.

There are four general elements. A *task* is the overall purpose of a program comprehension or development project. For example, "Instrument the Perl parser to access the abstract syntax tree" is a task. *Goals* often in the form of interrogative *questions* are used to gather relevant information about a program by framing comprehension as pieces of knowledge to discover. *Answers* to questions are found through the collection of artifact-based *evidence* or the direct derivation of answers by careful deliberation. Evidence gathered by the programmer includes information found in the program's code, documentation, or otherwise. Evidence occurs at various levels of abstraction, including the application, algorithmic, and code domains. Questions may take one of two forms: explicit and implicit. More common in the top-down process of program comprehension, explicit questions are premeditated and assume some sort of previous domain or program knowledge allowing the developer to form a reasonably specific question about a system. Implicit questions are formed as a result of gathering evidence, before asking any explicit questions, as in a bottom-up comprehension process. The implicit question usually is about the relevance of the particular piece of evidence collected, with a yes or no answer and a possible explanation. There may also be times when questions contain implicit hypotheses as to the question's answer or the hypotheses may be even hidden in the overall structure of the comprehension process.

The elements described above may be formed into a hierarchy of comprehension, similar to that described by Brooks in the top-down comprehension model [3]. Hierarchies can be considered a common-ground form of knowledge representation that is understandable by many, and not only the original author. In forming this hierarchy, tasks suggest the underlying purpose and are the reason for the existence of the knowledge construct. One or more primary comprehension goals are generated, sometimes framed by an implicit theory, to address the task. The primary goals may also be thought of as parts of or equivalent to the task. Generally, these primary goals cannot be directly tied to evidence, necessitating the creation of lower-level sub-goals, i.e., more concrete questions and testable hypotheses. The types of primary goals created depends on the nature of comprehension and the knowledge the developer already possesses. If the developer is familiar with the domain or program then more explicit questions may be asked before gathering evidence. However, if the developer is largely unfamiliar with the system, then evidence gathering and the creation of implicit questions takes place first. Through the gathering of evidence, questions are answered and hypotheses tested (supported or refuted). Higher-level goals are deemed successes or failures by aggregating the validity of lower-level sub-goals and hypotheses.

5.3 Motivation for Tool

The author's own experience in maintaining a software system that, at the time, was more than 12 years old, provided motivation to solve the problem of knowledge preservation in program design and comprehension. In this case, the project's age was the major deterrent to program comprehension as there was little to no formal documentation from the beginning of the project that would still be relevant to the current code base. Even comments inline with the code could be left out of date if a programmer forgot to update it after making a change. This information staleness would compound and propagate with each successive change to the code. Additionally, there was no central repository of knowledge or documentation for the system. The developer would have to piece together bits of information from change logs, bug reports, and word of mouth to get a good idea of some of the simplest concepts of the system. This was time consuming and often many gaps were left in the perceived comprehension. The cognitive load was not distributed evenly enough amongst humans and artifacts such that it was instead primarily placed on the human component.

Motivation for research into cognitive support in collaborative software engineering environments has already been described or hinted at in the previous chapters. Walenstein refers to cognitive support as the aid provided to humans in their thinking and problem solving by tools [45]. The highly collaborative nature of software development and current state of legacy software systems suggests that there is a need for a tool-driven methodology to properly document understanding as it occurs during the maintenance process or even as its own specialized documentation task.

Different Scenarios

There can be several different scenarios envisioned where a tool could be used to augment or replace disparate methods of documenting program comprehension.

Collaboration Aspect

In cases where the development team is made up of members that need to collaborate in order to accomplish software modification or program comprehension tasks, a tool that allows developers to work together and pool their work and resources would be of help. Even when direct collaboration does not seem necessary or is impeded by current methods, the distributed persistence of knowledge could open opportunities for the sharing of comprehension by reducing duplicative effort and fostering team cohesiveness.

Knowledge Preservation Aspect

In cases where the software system has a large lifetime over which team members may have come and gone, taking knowledge with them, it would be helpful to have a tool automatically persist documented knowledge so that it is accessible in the future. Also, team members may be dispersed over large distances or merely work in an environment where it is inconvenient to have rich face-to-face communications. Therefore, a tool that persists documented knowledge over a dispersed team would help to alleviate the disconnect between members who might otherwise use unreliable means of communication or find it a nuisance to communicate documented knowledge at all.

Integration of such a system into the development team's current environment is important, as adoption often relies on the convenience and unobtrusiveness of a tool that fits nicely into an existing workflow.

In our observational case study, we saw two programmers in a pair perform program comprehension tasks as part of their overall goal to instrument the code so that information about the program's internal state could be extracted for other purposes. Given an environment where the tools are an IDE and other typical items, a developer may use a paper notebook to document comprehension to themselves, leaving a missing link between their developed understanding and the artifacts of the software system. This was seen in the study as the subjects each took their own set of notes on their comprehension, parts of which were duplicative and others complementary.

A mechanism for collective knowledge preservation alone is not enough to ensure that comprehension and understanding is actually passed from one developer to the next. The representation of the knowledge should be something that most programmers would understand, but not so basic as to be entirely unstructured like a set of inconsistent notes. The construction and manipulation of the representation structure should also be flexible enough to accommodate different styles of documenting comprehension (e.g. bottom-up versus top-down). The style of documenting may depend on the specific situation of the individual developer and their relationship to the software system. Some possible permutations of such scenarios are shown below.

Imagine a set of scenarios where a task is given to a developer who:

a) has **both** domain and code knowledge
b) has some domain and some code knowledge
c) has domain, but no code knowledge
d) has code, but no domain knowledge
e) has **neither** code or domain knowledge (pure PC)

A developer in situation *a* can be considered an experienced expert who may not even need the assistance of documentation in order to complete a task. Instead they would rely on their own knowledge of both code and domain, but if necessary use a top-down comprehension approach. A developer in situation *b* would be less experienced and therefore rely on the use of documentation and a top-down approach to comprehension. Situation *c* would also likely result in a top-down process, as the developer would know enough about the domain to direct their questions and goals to specific areas of interest. Situation *e*, where the developer has very little to no knowledge of either code or domain, suggests the developer would use a bottom-up approach to comprehension because their initial work would be exploratory, both of the system and its domain. Situation *d* would seem to be a situation where the developer has only specific knowledge of the code, perhaps from working on utility or backend functionality, but no broad understanding of the domain to which the system is applied or how systems in the domain might typically be structured. In this case, either a top-down or bottom-up approach may be used depending on the particular task or piece(s) of the system under study, as if the developer is familiar with the some aspect of the task or code, they may use a top-down instead of bottom-up process. Situation *e*, with a developer lacking knowledge and documentation, is a typical case of where bottom-up comprehension would be used.

Maintenance of Context

Often, the interpretation or model of a developer's knowledge about a system or task is kept explicitly separate from the artifacts of the system itself, such as using notebooks to keep track of progress. This lends to excessive context switching as the developer reads and/or writes back and forth between the knowledge context and the system or code context. The *orientation* phase of program comprehension was observed in the study to have occurred multiple times during the course of each session. This indicated a loss of context, and was mostly seen to have occurred when the subjects were digging very deep within the code base, to very low levels of abstraction. At some point, the code is so low-level that it is difficult to see the overarching goal or question. In addition to the disadvantage of separating the knowledge store context from that of the system, when multiple developers are involved, there also exist multiple and disparate knowledge storage units which more often than not are only useful or interpretable to one person which relates back to the need for a commonly understood knowledge representation structure.

Having the mental context of comprehension maintained within software would reduce efforts in context switching, and hopefully also therefore reduce losses of context. The load of context switching may be further reduced by tightly integrating the documentation of comprehension within the developer's usual development environment as opposed to the use of an external tool.

5.4 Requirements

Based on our observations during the pair comprehension case study and the different scenarios of program comprehension discussed above, we describe a set of requirements for a cognitive support tool to aid the developer during the program comprehension process.

The observational case study conducted for this research also provides another source of data to motivate better tools and methods to support program comprehension.

One of the observed problems that occurred during the comprehension sessions was the frequent search through the various open windows on the computer screen to find the artifact relevant to the current goal in progress.

> R1: *A comprehension support tool should keep a connection between the goals and hypotheses of the developing cognitive model and the artifacts of the mental model [40] as this may improve the persistence of the programmer's workflow context.*

The usage of several different tools during the comprehension process also seemed to hinder the programmers as they seemed to need to reset their attention each time a switch occurred between tools.

> R2: *A comprehension support tool should be easily-adopted and unobtrusive, so that it is not jarring for a programmer when switching between different activities in the program comprehension process.*

The differences in note-taking structures between the observed programmers suggests that different pieces of knowledge are persisted for each participant and that this knowledge may or may not be retrievable in the future depending on the reader's interpretation of the notes and even whether or not the notes may be available.

> R3: *A comprehension support tool should provide a structure for the knowledge gleaned from a comprehension process, so that it is accessible by a larger audience, but also be flexible enough so that the programmer does not feel restricted in the kinds of knowledge he is able to represent.*

The persistence of knowledge throughout a team of developers is also important and such a tool should not confine the generated comprehension to the original programmer.

R4: *A collaborative comprehension support tool should disperse knowledge among the members of the team in real-time to maximize sharing of knowledge and minimize duplication of effort.*

Additionally, as communication during the observed comprehension sessions figured very prominently, we can see that pair comprehension does not occur within a vacuum.

R5: *A collaborative comprehension support tool should provide facilities for awareness of others working on the same or similar program comprehension tasks and also allow those users to easily communicate with each other through real-time synchronous methods if they are not co-located.*

The requirements described are similar to some of those proposed for collaborative reverse engineering tools by Ma [18].

We argue that these requirements may address or augment what Letovsky observed as the components of a knowledge-based program understander, which are [15]:

- **a knowledge base** encoding expertise and background knowledge
- **a mental model** encoding a programmer's current understanding
- **an assimilation process** interacting with the stimulus materials (target code and documentation) and knowledge base.

Our proposed tool keeps a database of previously built question-evidence hierarchies, a knowledge base of understanding. The process of building a comprehension hierarchy in the tool encodes a programmer's in-progress or current understanding of a system into a mental model. The assimilation process, as addressed here, involves the activities of gathering evidence in the tool and using the tool to answer questions and test hypotheses in conjunction with the use of knowledge already stored in the database.

5.5 Design and Implementation

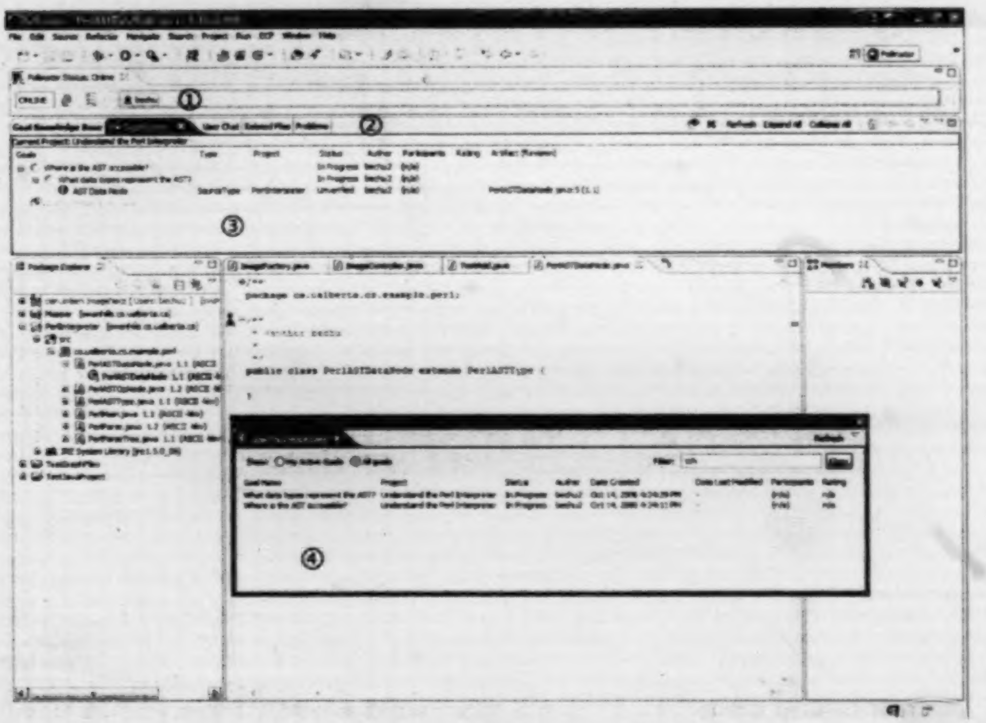


Figure 5.1: Screenshot of the *Pollinator* tool environment

A prototype Eclipse plug-in written in Java named *Pollinator* was created that integrates into the existing Java and plug-in development perspectives of the Eclipse IDE. The Eclipse open development platform is made up of several components targeted towards build, deployment, and management of software throughout its life cycle [1]. We are interested in using Eclipse as an integrated development environment (IDE) for the Java programming language. The Eclipse Java IDE, like most of the Eclipse project, is highly-extensible and composed of many plug-ins that can be extended or replaced by user-defined plug-ins.

The plug-in written as part of this thesis extends the base Eclipse Java IDE by implementing extensions to 14 extension points (see Appendix A.1 for details). Extension points are interfaces defined by plug-ins that allow extension or customization of the defining plug-in's functionality through the implementation of the interface or contract as an extension in another plug-in.

Pollinator contributes a perspective, views, actions, preferences, menus, annotations,

---

[1] http://www.eclipse.org/

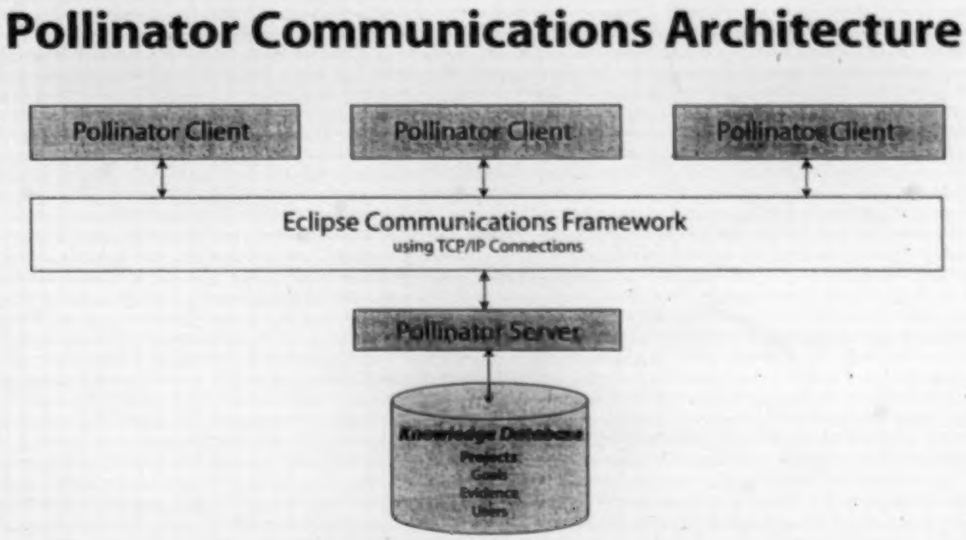

Pollinator Communications Architecture
Pollinator Client
Pollinator Client
Pollinator Client
Eclipse Communications Framework
using TCP/IP Connections
Pollinator Server
Knowledge Database
Projects
Goals
Evidence
Users

Figure 5.2: *Pollinator* communications architecture

decorations, and other extensions to the Eclipse Java IDE.

Collaboration features are implemented at a lower level through a simple protocol built on top of the Eclipse Communications Framework [2] (ECF) using a client-server architecture (see Figure 5.2). The use of the ECF as the base for the communications infrastructure allowed easier implementation of features like the sharing of knowledge and awareness amongst a group of users connected to a central server by abstracting away many of the low-level aspects of the communications infrastructure.

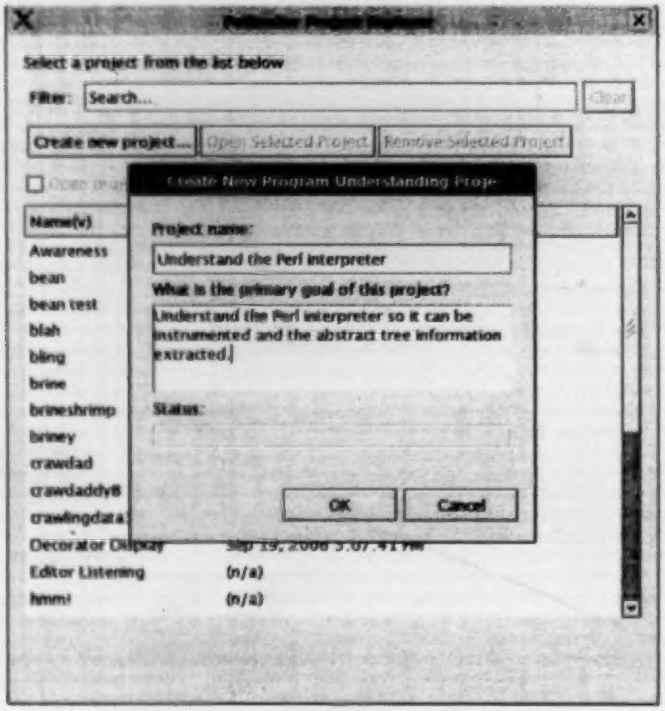
A screenshot of the *Pollinator* perspective in the Eclipse environment is shown in Figure 5.1.

The Eclipse IDE is organized into several components. Within each main workbench window is a workbench page that contains the visual presentation of the window's contents. The contents may generally be classified as one of two types of "parts", editors and views. A pre-defined arrangement and layout of editors and views is called a perspective. The Eclipse IDE defines several perspectives such as *Java*, *Plug-in Development*, and *CVS Repository Exploring*.

We defined a new perspective for our plug-in called *Pollinator* which has a layout similar to that of the existing *Java* perspective. The *Package Explorer* and *Navigator* views are placed to the left of the main source editor with the *Members* view on the right side. The *Pollinator* perspective also includes another view we implemented that stretches horizontally along the top of the window like a banner, providing passive peripheral user awareness. Just below the banner is a view folder (an Eclipse widget capable of holding many views in a tabbed layout) containing views for seeing and manipulating the documented program comprehension structure (comprehension modeling views).

---

[2] http://www.eclipse.org/ecf


Select a project from the list below
Filter:
Search...
Clear
Create new project...
Open Selected Project
Remove Selected Project
Create New Program Understanding Proje
Name(v)
Awareness
bean
bean test
blah
bling
brine
brineshrimp
briney
crawdad
crawdaddy8
crawlingdata
Decorator Display
Editor Listening
hmm!
(n/a)
(n/a)
Status:
OK
Cancel

Project name: Understand the Perl interpreter

What is the primary goal of this project?
Understand the Perl interpreter so it can be instrumented and the abstract tree information extracted.

Figure 5.3: *Pollinator* project explorer

Figure 5.4: Awareness view of *Pollinator* environment

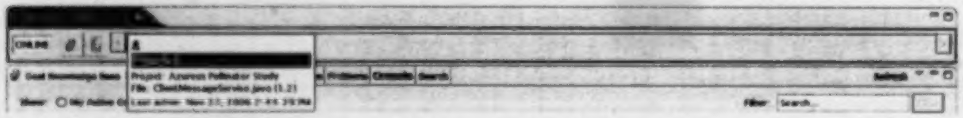


Figure 5.5: Team member detail in awareness view

The main awareness view (marked as 1 in Figure 5.1 and full-view shown in Figure 5.4) contains areas for program comprehension project manipulation and team awareness. The project explorer action displays a dialog (see Figure 5.3) that allows the opening, creation, and deletion of program comprehension projects on the *Pollinator* server. The projects referred to here are the equivalent of the *tasks* described in section 5.2. The team awareness area shows all users connected to the *Pollinator* server and displays a short status of the user's work context in a tooltip (see Figures 5.4 and 5.5). Clicking on a user brings up a context menu that allows actions to be performed, such as opening a user-to-user chat or viewing more detailed information about that team member. This awareness "banner" is somewhat similar to the concept of the Jazz Band in the Jazz collaboration project [4].

The auxiliary view folder, marked as **2** in Figure 5.1, contains several views meant to help with the cognitive support for the comprehension model of the system under study.

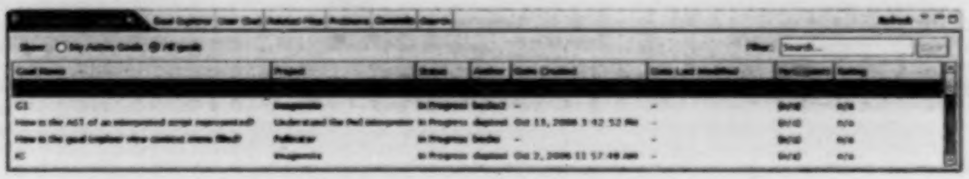


Figure 5.6: *Pollinator* knowledge base view

The knowledge base awareness view (detached view marked as **4** in Figure 5.1 and full-view shown in Figure 5.6) shows all of the currently constructed program understanding goals / questions (goals and questions are used interchangeably in this section) stored in all projects on the server. A text search allows the user to filter out questions of interest.

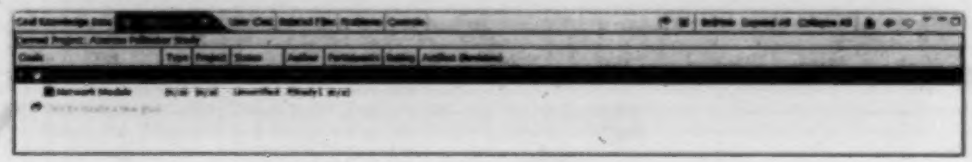


Figure 5.7: *Pollinator* goal explorer view

The comprehension manipulation view named "Goal Explorer" (marked as **3** in Figure 5.1 and full-view shown in Figure 5.7) allows the user to view and generate hierarchical models of understanding using goals and evidence. Currently users begin by creating one or more primary goals and then continue by generating sub-goals and questions through the process of top-down or bottom-up model building. As discussed earlier, the top-down model building process involves the asking of explicit questions and then gathering evidence or synthesizing answers in order to resolve the question. The bottom-up process starts with generalized goals and questions, which are not directly answered by the gathering of evidence, but these pieces of evidence are implicit questions in their own right. The implicit question here is: is this evidence relevant to this question? Users gather evidence through several methods. One method is the dragging of elements from Java source code views in the Eclipse environment (such as the *Package Explorer* and *Members* views seen in Figure 5.11) onto goal / question nodes of the *Goal Explorer* tree. This method acts as a kind of bookmarking or annotation mechanism for the pieces of evidence gathered so that they may be referred to later in the process of question resolution. Each piece of evidence dragged using this method is tied to a specific version of the source code in a version control system (currently CVS) so that the state and context of the artifact is preserved for later interpretation. A second method is the explicit attaching of external evidence to a question in the form of a URL, a saved chat, a file, or simply free-form text. The different types of evidence are distinguished by different icon types and are represented differently according to their underlying nature. Goals and evidence have states associated with them to track progress. For example, goals have the states of *in progress*, *answered*, and *failed*. Evidence may be marked as *relevant*, *irrelevant*, or *unknown*. Within the "Goal Explorer", the question-evidence hierarchy is reasonably flexible, allow the user to add and delete goals and evidence at any point in the hierarchy. The user may also drag and drop parts of the hierarchy to any other place in the hierarchy if such a rearrangement is needed.

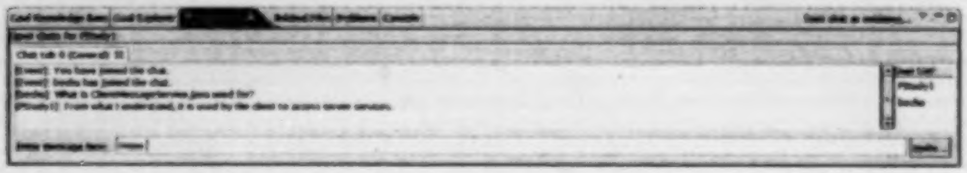


Figure 5.8: *Pollinator* user chat view

The "User Chat" view (see Figure 5.8) allows direct communication between members of a development team through textual chat, either one-on-one or as a group with members added by invitation. The text of the chats, as mentioned above, may be saved as evidence if desired.

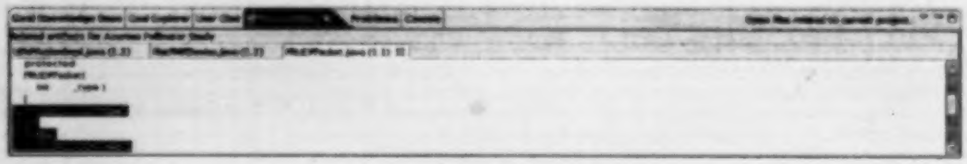
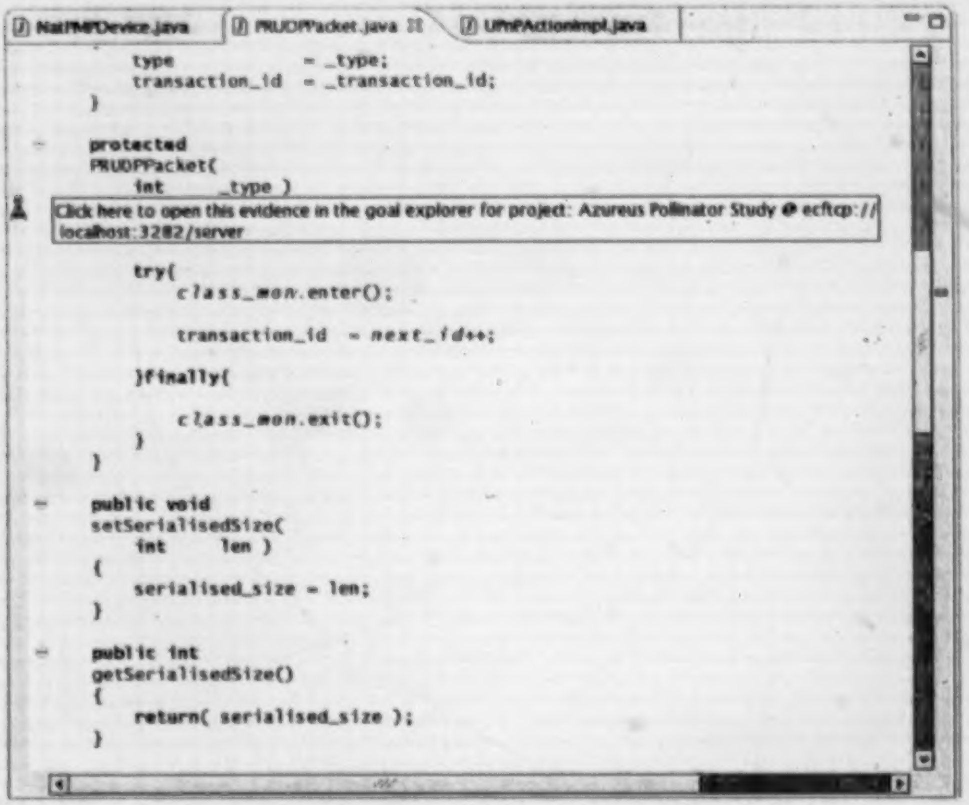


Figure 5.9: *Pollinator* related files view

The "Related Files" view (see Figure 5.9) collects all of the file-based evidence gathered for a particular project in one tab view. *This makes it easy to see at a glance, which parts of the source code base are involved in a particular program comprehension project.*



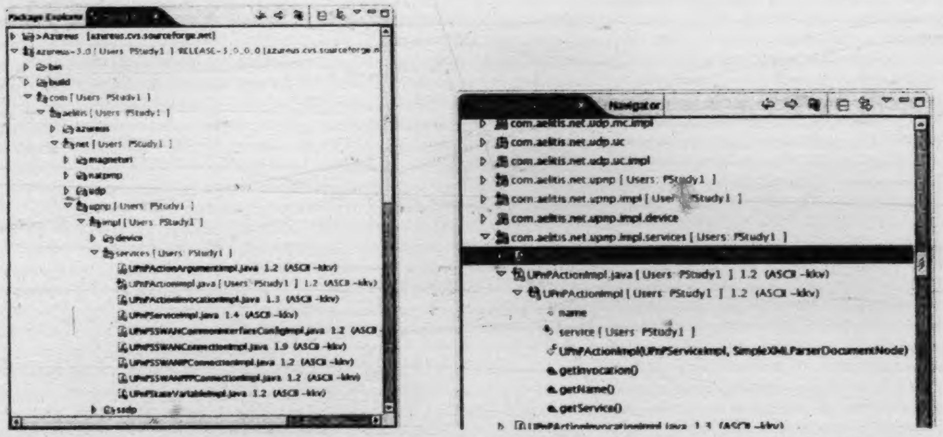
```
        type            = _type;
        transaction_id  = _transaction_id;
    }

    protected
    PRUDPPacket(
        int      _type )
```

Click here to open this evidence in the goal explorer for project: Azureus Pollinator Study @ ecftcp://localhost:3282/server

```
        try{
            class_mon.enter();

            transaction_id  = next_id++;

        }finally{

            class_mon.exit();
        }
    }

    public void
    setSerialisedSize(
        int     len )
    {
        serialised_size = len;
    }

    public int
    getSerialisedSize()
    {
        return( serialised_size );
    }
```

Figure 5.10: *Pollinator* editor annotation

There are also many different types of awareness that *Pollinator* incorporates into the Eclipse environment. If the currently active editor has knowledge associated with it in the database, an annotation marker is placed in the left vertical ruler that links back to the project and details of the comprehension documented about the file (see Figure 5.10). To facilitate peripheral awareness of team member activity, *Pollinator* also decorates relevant views of the Eclipse environment to denote the artifacts other team members may be working on. For example, if a user called "PStudy1" has an editor open on the file `UPnpActionImpl.java`, the *Navigator* view in Eclipse will decorate that item and its parent artifacts in the tree view, as seen in Figure 5.11a. If the user is looking at a *Package Explorer* view, then they may also see the particular method or other data structure the user "PStudy1" is examining (see Figure 5.11b). The *Members* view is also decorated with this peripheral awareness information (see Figure 5.11c). This provision of periph-

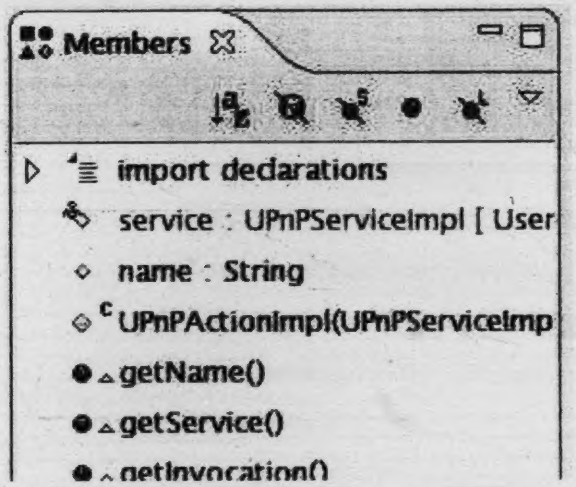


```
▷ >Azureus [azureus.cvs.sourceforge.net]
▽ azureus-3.0 [ Users PStudy1 ] RELEASE-3_0_0_0 [azureus.cvs.sourceforge.n
  ▷ bin
  ▷ build
  ▽ com [ Users PStudy1 ]
    ▽ aelitis [ Users PStudy1 ]
      ▷ azureus
      ▽ net [ Users PStudy1 ]
        ▷ magneturi
        ▷ natpmp
        ▷ udp
        ▽ upnp [ Users PStudy1 ]
          ▽ impl [ Users PStudy1 ]
            ▷ device
            ▽ services [ Users PStudy1 ]
                UPnPActionArgumentImpl.java 1.2 (ASCII -kkv)
                UPnPActionImpl.java [ Users PStudy1 ] 1.2 (ASCII -kkv)
                UPnPActionInvocationImpl.java 1.3 (ASCII -kkv)
                UPnPServiceImpl.java 1.4 (ASCII -kkv)
                UPnPSSWANCommonInterfaceConfigImpl.java 1.2 (ASCII
                UPnPSSWANConnectionImpl.java 1.9 (ASCII -kkv)
                UPnPSSWANIPConnectionImpl.java 1.2 (ASCII -kkv)
                UPnPSSWANPPPConnectionImpl.java 1.2 (ASCII -kkv)
                UPnPStateVariableImpl.java 1.2 (ASCII -kkv)
          ▷ ssdp
```

(a) *Pollinator* decorations in the *Navigator* view of Eclipse.

```
▷ com.aelitis.net.udp.mc.impl
▷ com.aelitis.net.udp.uc
▷ com.aelitis.net.udp.uc.impl
▷ com.aelitis.net.upnp [ Users: PStudy1 ]
▷ com.aelitis.net.upnp.impl [ Users: PStudy1 ]
▷ com.aelitis.net.upnp.impl.device
▽ com.aelitis.net.upnp.impl.services [ Users: PStudy1 ]

  ▽ UPnPActionImpl.java [ Users: PStudy1 ] 1.2 (ASCII -kkv)
    ▽ UPnPActionImpl [ Users: PStudy1 ] 1.2 (ASCII -kkv)
        name
        service [ Users: PStudy1 ]
        UPnPActionImpl(UPnPServiceImpl, SimpleXMLParserDocumentNode)
        getInvocation()
        getName()
        getService()
```

(b) *Pollinator* decorations in the *Package Explorer* view of Eclipse.



```
▷ import declarations
  service : UPnPServiceImpl [ User
◇ name : String
◇ UPnPActionImpl(UPnPServiceImp
● getName()
● getService()
● getInvocation()
```

(c) *Pollinator* decorations in the *Members* view of Eclipse.

Figure 5.11: *Pollinator* decorations in Eclipse

eral awareness through decorators is similar to that provided by the Jazz project's "Concert Awareness" feature [4].

The tool persists several pieces of knowledge about each goal and question, such as the user who created them and the understanding project they are associated with. Additionally, versioning information on the evidence associated with hypotheses is stored if possible. This allows a programmer to go back to the specific artifact used in a comprehension model and also maintains a connection between the cognitive and program models.

We believe *Pollinator* is flexible enough to be used in both bottom-up and top-down comprehension processes, as observed in the case study and based on the following methods of use.

In hypothesis-driven top-down comprehension, several goals and questions are stated first and code is looked at last. *Pollinator* might be used in such a situation as follows:

1. *Create new task with explicit goal*

   e.g.

   Task: Understand the Perl Interpreter

   Description: Understand the Perl interpreter so it can be instrumented and the abstract syntax tree information extracted.

2. *Create new explicit question or goal*

   e.g.

   Question: Where is the AST accessible?

3. *Create a sub-goal that is a concrete, explicit question*

   e.g.

   Question: Which data types represent the AST?

4. *Drag and drop evidence to support an answer to question*

   e.g.

   Drag the files `ASTNode.java` and `ASTOp.java` to the just-created question.

In code-driven bottom-up comprehension, there is no initial specific goal and the behavior seems mainly exploratory. Therefore, the initial activity is almost one of "bookmarking" the code. Later on, these bookmarks can be organized in a manner that creates or answers more specific questions, which in turn requires the gathering of more evidence. *Pollinator* might be used in such a situation as follows, as relating to the previous top-down example:

1. *Create new task with non-specific, exploratory goal*

   e.g.

   Task: Understand the Perl interpreter

   Description: Find out how the interpreter works.

2. *Create an exploratory goal from which to begin*

   e.g.

   Goal: What makes up the Perl interpreter?

3. *Add all exploratory evidence (i.e. bookmark points of interest)*

   e.g.

   Drag the files `PerlIMain.java`, `PerlParse.java`, `Util.java`, `ASTNode.java`, `AST.java`, `IOCache.java`, etc. to the just-created goal.

4. *Later, organize "buckets" of evidence by creating new questions / goals that group together the gathered evidence.*

   e.g.

   New Question: What modules make up the support functionality that is not directly related to parsing?

   Then group `Util.java` and `IOCache.java` under this question.

We believe *Pollinator* shows promise in initial uses of the tool to build and maintain the tool itself. The primary author found himself preferring the tool to a vanilla Eclipse software development environment as it allowed the higher-level structure of the program to emerge without interfering with the usual workflow. A more thorough evaluation of the tool follows in the next two chapters.

Chapter 6

Preliminary Evaluation

> “Reverse engineering techniques should not be simply inserted into the software process, but that they should be tightly integrated with the entire process... reverse engineering tools should be supported by well-established means of team communication” - van Deursen

6.1 Methodology

We present a preliminary evaluation of Pollinator by its feature set in fulfillment of the derived tool requirements, support for cognitive aspects of program comprehension and collaboration in comparison to related tools, and its usability and viability in a small heuristic evaluation and cognitive walkthrough of an experimental usage scenario.

6.2 Feature Set

Pollinator fully addresses a number of the requirements specified in Section 5.4 and all of them to at least some degree. The requirements are listed again here for reference.

> R1: *A comprehension support tool should keep a connection between the goals and hypotheses of the developing cognitive model and the artifacts of the mental model [40] as this may improve the persistence of the programmer's workflow context.*

Pollinator addresses this through the explicit hierarchical representation of goals, questions, answers, and evidence in the “Goal Explorer” view. The connection is maintained through manipulation and exploration of this comprehension structure, and there is always an explicit piece of data that represents some understanding or comprehension about the software system. The persistence of knowledge is accomplished by storing documented comprehension in a database. In Walenstein's theory-directed cognitive support analysis, the question-evidence hierarchy may be considered an external memory for storing a “plan” [45]. The model is connected to the source code through editor annotations, view decorations, and the related files feature. The knowledge base provides an overview of all the documented comprehension for a software system, providing answers to questions that were previously asked and perhaps reducing duplicative work effort.

> R2: *A comprehension support tool should be easily-adopted and unobtrusive, so that it is not jarring for a programmer when switching between different activities in the program comprehension process.*

Requirement R2 is partially addressed by the integration of Pollinator into an existing development environment, Eclipse, and also through the reuse of views used in Java software development. The tool adapts the existing Java development perspective, so the amount of context switching should be decreased as compared to using an external tool or a tool that requires the use of its own Eclipse perspective. The initial configuration of the tool was more obtrusive than the current layout as there was a larger portion of the screen devoted to the user awareness and project information. "Goal Explorer" is still an awkward occupant of screen real-estate, however efforts to reduce its obtrusiveness have been partly hampered by the restrictions on perspective layout in the Eclipse development environment [1].

As Pollinator is implemented as an Eclipse plug-in, we may take advantage of the pre-existing deployment functionalities available through Eclipse's export features to reduce the burden in adopting Pollinator as a tool. The server component may be easily deployed on a single computer while the client plug-in is simply copied to the existing location of Eclipse on the developer computers. Future work in this area may include making use of so-called *zero-configuration* network protocols, such as Apple's *Bonjour*[2], so clients may automatically discover a Pollinator server without having to manually enter this information.

> R3: *A comprehension support tool should provide a structure for the knowledge gleaned from a comprehension process, so that it is accessible by a larger audience, but also be flexible enough so that the programmer does not feel restricted in the kinds of knowledge he is able to represent.*

The comprehension trees provide for a minimal amount of defined structure (hierarchical form, questions supported by answers and evidence) and are flexible enough to accommodate different styles of comprehension. These hierarchies are also accessible to a larger audience through the idea of framing the data as a knowledge base of frequently asked questions and answers, linked to the code base and other artifacts.

> R4: *A collaborative comprehension support tool should disperse knowledge among the members of the team in real-time to maximize sharing of knowledge and minimize duplication of effort.*

---

[1] Bug 151715 (Allow More Programmer Control over Perspective Layout): https://bugs.eclipse.org/bugs/show_bug?id=151715

[2] http://developer.apple.com/networking/bonjour/

Pollinator disperses changes in program understanding to the entire team in real-time. Duplication of effort is minimized through the sharing or publishing of comprehension projects in the "Project Explorer", the changes made to the comprehension model in individual projects in "Goal Explorer", and the "global" sharing of comprehension goals and questions through the "Knowledge Base" view.

The understandings collected are persisted in a database to be accessed at any future point in time, in addition to being immediately persisted to every client of the Pollinator system. The understandings are also connected to the artifacts according to available versioning information, so that the relevance of the documentation is preserved. In this way knowledge is persisted over time and space.

> R5: *A collaborative comprehension support tool should provide facilities for awareness of others working on the same or similar program comprehension tasks and also allow those users to easily communicate with each other through real-time asynchronous methods if they are not co-located.*

Pollinator implements passive awareness of other users by showing their connected state, the files and source-level structures being viewed, and last observed activity. The banner view provides at-a-glance information on who else is connected to the Pollinator system. Through tooltips, the developer may also see what project the team member is working on, their activity status, and which file they may have open. File use information is also shown by the decorating of various information views in Eclipse, such as the package explorer and members views with text denoting what other members are viewing. Developers are also aware of the work of others through the shared listing of goal/question authors and participants in the knowledge base and goal explorer views.

Passive awareness is also provided through the real-time population of the various comprehension views initiated by events received from other users and the Pollinator server, as described for requirement R4.

Active awareness and communication is facilitated in Pollinator by real-time asynchronous single and group user chats (i.e. instant messaging) which may be initiated either through explicit member-to-member contact or through a team member's association to a particular system artifact, such as a viewed source code file. The chats may be supplemented by regular face-to-face meetings and email if necessary.

The list of requirements for collaboration in software maintenance tasks described in Chapter 3 can also be used as a metric against which to compare Pollinator:

1. *Maintain a dialog with team members*

   This is accomplished through the mechanisms of passive and active awareness of Pollinator. Team member dialog may be explicit, through group user instant messaging, or implicit through the passive awareness of activities of team members.

2. *Allow the team to simultaneously work on a single task*

   Parallel work is possible, and in fact encouraged by the tool by emphasizing the sharing of knowledge and allowing users to work on program comprehension projects simultaneously.

3. *Send messages/e-mail to team members*

   Again, active communication in the tool is facilitated through instant messaging. Integrating e-mail into the tool may be considered future work, perhaps through making use of the `mailto:` protocol so that a developer's preferred email client is actually used.

4. *Concurrent use of dictionary and diagrams*

   In order to apply this requirement to Pollinator, we may consider a dictionary to be the "Knowledge Base" of our tool. The concurrent manipulation of this dictionary is indeed possible through the creation, deletion, and modification of goals and questions.

5. *Group interaction support (e.g. brainstorming)*

   Collective communication is possible through the group chat mechanism, facilitating textual brainstorming.

6. *Attach electronic notes to objects*

   In the case of Pollinator, electronic notes may be attached to objects in the form of annotating source code and other file artifacts. This links the artifacts as pieces of evidence to program comprehension models.

7. *Anonymous feedback or input*

   This requirement is not considered for Pollinator, as it may not be applicable.

8. *Notify engineers if a design change affects their work*

As Pollinator is meant primarily as a tool for documenting program comprehension, design change notification would probably take the form of redocumenting the comprehension and dispersing that changed information to the rest of the team.

9. *Build a catalog of macros accessible by the team*

   This requirement is not considered for Pollinator, as it may not be applicable.

	Pollinator	FEAT/ConcernMapper	JASPER	Mylar	JAZZ
Focus/Target	Documenting program comprehension in collaborative environments	Abstract low-level source code elements into functional units	Abstract artifacts of software system into task contexts, reduce redundant navigation	Focus the current task context through filtering, reduce cognitive work load	Collaborative software development environment
Cognitive Support	Question-evidence hierarchy knowledge representation, maintain connection between mental model and source code.	Use of concerns as abstractions of software, grouping functionality spread over the system	Similar to idea of concerns, except use idea of working sets of artifacts and show relevant views to create task context	Filtered task context by automatically analyzing artifact degrees of interest and only showing relevant objects to the user	Collaboration awareness, supplement the agile development process
Collaboration Support	Passive and active team awareness	Save and loading of concern files	Working sets may be archived in version control	Sharing and synchronization of tasks through Bugzilla integration	Passive peripheral and active team awareness, multiple teams, screen sharing
Obtrusiveness Relative To Pollinator	Medium	High	High	Low (automation)	Medium

Table 6.1: Comparative summary of Pollinator and related tools

6.3 Tool Comparison

While the related work chapter of this thesis contains brief summaries of the related tools and a few minor critiques, this section takes a more detailed view of Pollinator compared to the related work, in light of the in-depth discussion of its implementation in the previous chapter.

Table 6.1 shows a comparative summary of four related tools and Pollinator. After looking at the features and focus of related work, we found that Pollinator is the only tool to emphasize both cognitive and collaborative support in its capabilities. FEAT/ConcernMapper, JASPER, and Mylar are primarily concerned with the cognitive support of tasks related to program comprehension. JAZZ is a more general framework that focuses on providing general collaborative abilities, but does not design for anything specifically related to cognitively supporting the documentation of program comprehension.

When looking at the capabilities of the primarily program comprehension-related cognitive support tools (FEAT, ConcernMapper, JASPER, and Mylar) we found both areas of strength and weakness in relation to our tool. In FEAT/ConcernMapper, its strength is on emphasizing the abstraction of scattered source elements into more meaningful representations as software concerns. The obtrusiveness of tool, however, seems high compared to Pollinator as it tends to require several views to be opened at once, obscuring much of the normal developer workflow, whereas Pollinator tries to preserve as much of the typical development perspective as possible. JASPER has both similar strengths and weaknesses. Rather than representing the software concerns in a graph format, JASPER groups together "working sets" of source code, essentially snippets of code that taken together represent some abstract functionality. It also, however, relies on the use of an obtrusive interface that obscures the normal development views and perhaps requires a fundamental change in the way the development process works. Of these tools, Mylar seems the most unobtrusive tool perhaps because its focus is on reducing the cognitive workload by filtering the software system and not abstracting it as is the case for Pollinator and the other related work. Mylar also excels in its integration of a learning system that constructs its filtering schema by observing the concrete actions and behaviors of the user, thereby reducing the amount of proactive effort the user must put in to take advantage of the tool.

The collaboration aspects of Pollinator may only fairly be compared against those of a tool designed with this in mind. It does not appear that any of the cognitive support tools discussed above focus on collaboration and team awareness. For example, one of the primary concerns of Pollinator is the sharing of knowledge amongst the members of a team easily and immediately. Of the tools that do permit the persistence of knowledge, the mechanism relies on the manual saving and restoration of files containing the data, with no evident means to communicate the data to others from within the tool itself. JAZZ, however, is a development environment that is especially concerned with supporting collaboration awareness to support the agile software development process. Therefore, its cognitive support focus is not on reducing efforts related to program comprehension, but rather on strengthening the collaborative ties between members of a team and other stakeholders in the entire engineering process. In this way, it features a rich set of collaborative features including not only passive and active peripheral team awareness, but also support for multiple teams, screen sharing, discussion boards, team event logs and other management-related tools. In comparison, Pollinator's collaborative features assume there is already an adequate management structure in place, and focuses on providing team support for the process of developing and sharing program comprehension knowledge and documentation.

Pollinator's general weakness amongst the tools, is currently its obtrusiveness and degree of integration with Eclipse. The obtrusiveness stems from the necessity of the user to manually create the knowledge representation structure, whereas Mylar and to some degree JASPER, take into consideration how tool usage might be made less intrusive through automation or novel interaction techniques, such as mouse gestures. While we believe Pollinator provides a moderate degree of seamless integration with Eclipse, its client-server nature means that there is a disconnect between starting up Eclipse and connecting to the program comprehension server. This is alleviated somewhat through the provision of autologin functionality that remembers your state and project from the last session. A tool like Mylar, which is focused on providing a context for task development, actually modifies more of Eclipse to suit its purpose, such as altering view contents and relating them to the tasks more seamlessly.

6.4 Heuristic Evaluation of Usability

Nielsen suggests a heuristic evaluation of user interfaces to judge usability [23]. A user interface is evaluated by a set of guidelines or "rules of thumb" that are non-specific, and therefore heuristic in nature. Nielsen recommends three to five evaluators at the most, however we conduct the evaluation using only one evaluator due to time constraints. Each evaluator goes through the interface at least twice, inspecting various "dialogue elements" and compares them with a list of recognized usability principles (the heuristics). The heuristics are general rules that should describe common properties of usable interfaces. In the general case, an evaluator may also consider any other additional principle of usability that they see fit to apply to the interface or any specific dialogue element.

The main difference between heuristic evaluation and traditional user testing is that in heuristic evaluation, the onus for analyzing the user interface is on the evaluator, whereas in traditional user testing the evaluator's actions are interpreted by an observer or experimenter, so that they may be related to possible usability issues in the user interface.

Nielsen has developed a list of ten usability heuristics that can be used to evaluate user interface design [24]. The list was originally created in 1990 and has been refined in the years since. We use Nielsen's heuristics (shown below in italics) to conduct a small-scale heuristic evaluation of *Pollinator's* user interface.

- *Visibility of system status*
  The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

  > Through the use of the *Pollinator* status bar at the top of the perspective as well as the decorators and annotations present throughout the interface, the plug-in keeps the user informed about the current state of the team and documented comprehension. Feedback is given within reasonable time due to the instantaneous nature of the peripheral awareness and knowledge propagation mechanisms.

- *Match between system and the real world*
  The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

  > The nomenclature of the documented knowledge should be familiar to almost anybody, due to the words' usage being commonplace in everyday English. For example, the words "question" and "evidence" have general meanings that can be applied in many situations, and in the case of program comprehension their meaning should be easily grasped. However, some may be confused with the idea of "evidence", because there is no explicit separation between evidence and a question's answer.

- *User control and freedom*

  Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialog. Support undo and redo.

  > The system does not explicitly support the undo/redo functionality since the manipulation of the data structures automatically persists the changes immediately. However, the operations are simple enough and there are confirmation dialogs for destructive actions that there is sufficient user freedom and control.

- *Consistency and standards*

  Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

  > The plug-in, by necessity, adheres to many Eclipse conventions of use and therefore should not be confusing to the user.

- *Error prevention*

  Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

  > There are several situations in which Pollinator prevents the user from committing an invalid action (i.e. adding an annotation to a file that has not been updated from CVS) or at least warns that the action may have certain negative consequences so that the user may decide the ultimate course of action.

- *Recognition rather than recall*

  Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialog to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

  > The layout of the perspective is static, unless manually altered by the user, and the tabbed nature of the auxiliary view folder allows for fast switching

between the different views without losing overall context. The peripheral awareness view is always visible to the user.

- *Flexibility and efficiency of use*
  Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

  The use of accelerators is very sparse, so this may be a deficiency of the system. However, there are accelerators for perceived common actions such as creating a goal and adding a file as evidence.

- *Aesthetic and minimalist design*
  Dialogs should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialog competes with the relevant units of information and diminishes their relative visibility.

  The dialogs only contain enough material to complete the task for which they were designed.

- *Help users recognize, diagnose, and recover from errors*
  Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

  If errors do arise, they are presented in plain English and where possible, a workaround or reason for the error is also displayed.

- *Help and documentation*
  Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

  There is no online help system or documentation available for the Pollinator system.

6.5 Cognitive Walkthrough of Usability

Walkthroughs are an alternative to heuristic evaluations for predicting users' problems without doing user testing [28]. The basic idea is to walk through performing a task with the tested system, noting problematic usability features at each step.

In particular, a cognitive walkthrough is meant to simulate a user's problem solving process at each step in the dialog of human-computer interaction [25]. The test is whether or not a user's goals and memory for actions can be assumed to lead to a next correct action.

The steps of cognitive walkthrough as illustrated by Preece, are as follows:

1. Identify and document characteristics of typical users, and develop sample tasks focusing on aspects of design to be evaluated.

2. The designer and one or more expert evaluators come together to do analysis.

3. Evaluators walk through action sequences for each task, placing them within context of a typical scenario, while trying to answer the following questions:

    - Will the correct action be sufficiently evident to user?
    - Will the user notice that the correct action is available?
    - Will the user associate and interpret the response from an action correctly?

    Basically, will users know *what* to do, see *how* to do it, and understand from feedback whether the action taken was correct or not.

4. A record is compiled as the walkthrough is being done, including critical information in which:

    - assumptions about what would cause problems and why are recorded, including explaining why users would face difficulties.
    - notes about side issues and design changes are made.
    - a summary of results is compiled.

5. The design is then revised to fix problems presented.

Some important points to keep in mind before, during, and after a cognitive walkthrough include keeping account of what does and does not work, using a standardized feedback form that directs the evaluators to answer the above three questions at each step and record details outlined in the first four points in addition to the date of the evaluation, documenting details of the software version and the evaluators' names, and documenting the severity of the problems.

The strengths of a cognitive walkthrough are that it focuses on users' problems in detail, but does not require the presence of real users nor a working prototype. However, these

Attribute	Value
Language	Java
Version	2.5.0-CVSBeta
Lines of Code	258,998 [a]
Number of Classes	1843
Number of Interfaces	664
Number of Packages	373
Number of Developers	31 (13 active)[b]
Age of Project	3+ years (June 2003)[c]

[a] As measured by Metrics: http://metrics.sourceforge.net/

[b] Azureus Ohloh Metrics Report: http://www.ohloh.net/projects/84

[c] Azureus Wikipedia entry: http://en.wikipedia.org/wiki/Azureus

Table 6.2: Open-source project Azureus at a glance

walkthroughs may be very time-consuming and laborious while their narrow focus may be useful for certain types of systems but not others.

A small experimental test to demonstrate the use of Pollinator and provide an initial evaluation of its abilities was undertaken. The purpose of the study was to take an existing project fitting the profile of a large and complex software system that has had developer turnover over a period measurable in years. Azureus[3], an open source BitTorrent[4] peer-to-peer file distribution client/server written in Java, was chosen as the subject system.

Azureus is well-suited to be the subject of a case study for Pollinator. Its characteristics fit the profile of the type of a large software system that may require cognitive support in documenting program comprehension. As can be seen from Table 6.2 the codebase is large enough to qualify as an example of PitL (programming in the large) according to Holt, as the number of lines of code exceeds 250,000 [11]. Furthermore, according to ohloh.net, the Azureus source code has a low number of comments relative to other Java projects [5]. This indicates that there may be ample opportunity for documentation to support comprehension of the source code due to the current lack of inline commenting on such a complicated project, assuming there is also little external documentation on Azureus. In terms of team members, ohloh.net also states that while there have been 31 developers over the project's lifetime only 13 are currently active, suggesting developer turnover that may have led to a loss of team knowledge.

For the study, we undertook a task that might be done as part of software maintenance, a small extension to the existing application functionality. The knowledge we had prior to beginning the task was strictly the experience we obtained from regular use of Azureus as an application; the code base and general software architecture was unfamiliar to us. Therefore, this could be classified as a scenario where there was domain knowledge, but very little to no code knowledge.

---

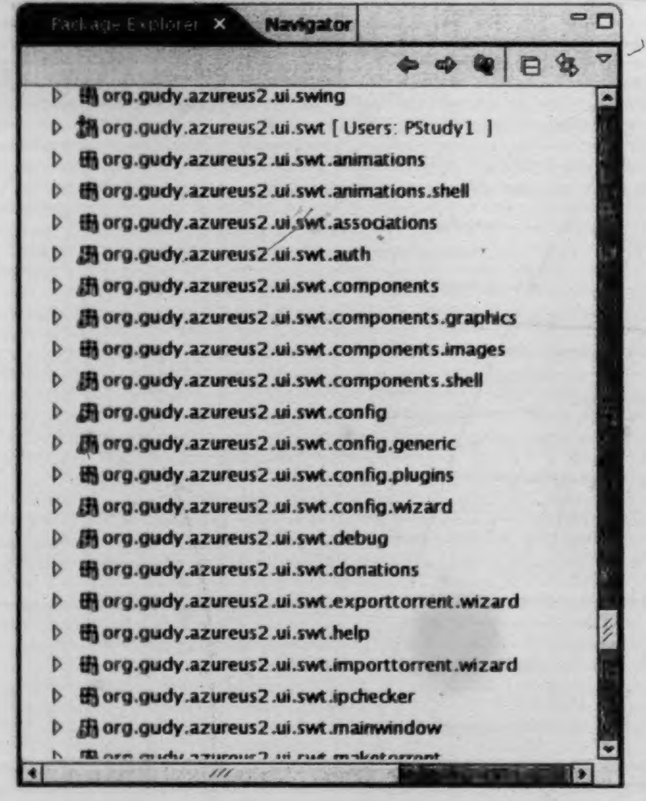
[3] http://azureus.sourceforge.net/

[4] http://en.wikipedia.org/wiki/BitTorrent

[5] Ohloh Azureus Metrics: http://www.ohloh.net/projects/84

The task was to extend Azureus with a function to restart torrent downloads by an action through the context menu of a selected torrent. Currently, there are separate stop and queue/force start functions, but no single restart function. During the task, we used Pollinator to document the comprehension gained along the way.

Since the situation was one of no code knowledge and some domain knowledge, the comprehension process followed could best be described as mainly bottom-up in nature. However, since there is some domain knowledge of the application and the general architecture of Java applications, the initial goal was not purely exploratory.



- org.gudy.azureus2.ui.swing
- org.gudy.azureus2.ui.swt [Users: PStudy1]
- org.gudy.azureus2.ui.swt.animations
- org.gudy.azureus2.ui.swt.animations.shell
- org.gudy.azureus2.ui.swt.associations
- org.gudy.azureus2.ui.swt.auth
- org.gudy.azureus2.ui.swt.components
- org.gudy.azureus2.ui.swt.components.graphics
- org.gudy.azureus2.ui.swt.components.images
- org.gudy.azureus2.ui.swt.components.shell
- org.gudy.azureus2.ui.swt.config
- org.gudy.azureus2.ui.swt.config.generic
- org.gudy.azureus2.ui.swt.config.plugins
- org.gudy.azureus2.ui.swt.config.wizard
- org.gudy.azureus2.ui.swt.debug
- org.gudy.azureus2.ui.swt.donations
- org.gudy.azureus2.ui.swt.exporttorrent.wizard
- org.gudy.azureus2.ui.swt.help
- org.gudy.azureus2.ui.swt.importtorrent.wizard
- org.gudy.azureus2.ui.swt.ipchecker
- org.gudy.azureus2.ui.swt.mainwindow
- org.gudy.azureus2.ui.swt.maketorrent

Figure 6.1: A portion of the many Java packages in the Azureus project

Here is an outline of the session, with cognitive walkthrough questions interspersed as questions labeled *Q* and answers in decreasing order of likelihood labeled *Ax* where *x* = *1*, *2, 3, …:*

1. With the Azureus source code checked out of the project's CVS repository into an Eclipse workspace, open the Pollinator perspective and connect to the server.

   Q: Will users know what to do?

   A1: Yes – they know in Eclipse that there is a way of opening different perspectives to activate relevant views. They also know that the functionality of Pollinator is not activated if not connected to a server since the views and buttons are disabled and messages of "Not connected" and "OFFLINE" are displayed in the banner view.

   A2: Yes – the user may already have the Pollinator perspective open and be configured to automatically connect to the server.

   A3: No – the user may not be aware that connecting to the server is necessary for using Pollinator.

   Q: Will users see how to do it?

   A1: Yes – the only active button will be the one to connect to the server and there is also a tooltip to explain each button's functionality.

   Q: Will users understand from feedback whether the action was correct or not?

   A1: Yes – once the connect button is pressed, the user will be either prompted to enter connection information or will automatically be connected to the server using previously-saved or pre-set settings thereby causing the Pollinator views to become active and populated. The banner view will change to show a status of "ONLINE" and also display any other users on the system.

2. Create a new program understanding project: *"Azureus torrent menu action addition"*

   Q: Will users know what to do?

   A1: Yes – the user will be unable to use the *Goal Explorer* view until creating a project as the message "No project open." is displayed, which should prompt them to create or open a project.

   Q: Will users see how to do it?

   A1: Yes – the banner view contains "global" Pollinator actions that are always in view of the user, including the *Project Explorer* button that allows users to create or open projects.

   A2: No – since the *Goal Explorer* shows the message "No project open." the user may expect the ability to create or open a project from within that particular view and not through the route of *Project Explorer*. The message could be augmented with an instruction like "Open the Project Explorer to create or open a project.".

Q: Will users understand from feedback whether the action was correct or not?
A1: Yes – once inside the *Project Explorer*, the creation or opening of a project results in that project becoming shown and focused in the *Goal Explorer* view.

3. Create primary question: *"Where are the torrent menu actions declared and defined?"*

   Q: Will users know what to do?
   A1: Yes – the *Goal Explorer* remains empty if there is a project with no goals and a project with goals will not progress if no new goals are added.

   Q: Will users see how to do it?
   A1: Yes – the action to create a new goal is made obvious through the use of an "in-place" cue for a user to click in the area where a goal should appear with the text "Click to create new goal". There is also a toolbar button that can be used to create a new goal, with a fairly intuitive icon and tooltip text.

   Q: Will users understand from feedback whether the action was correct or not?
   A1: Yes – upon clicking the cue or the button, the user will be presented with a dialog to create a new goal. This dialog is presented in a format that should be familiar to most users who have experience filling out computerized forms.

4. Gather evidence:

   - In Eclipse's Package Explorer browse through the packages looking for one that might be a good candidate for containing user interface functionality (see Figure 6.1).
   - Look through the `org.gudy.azureus2.ui.swt.mainwindow` package for candidate classes where the context menu actions might be declared.
   - Decided that `MainWindow.java` was probably a good place to start and dragged the file into the primary question.

     Q: Will users know what to do?
     A1: No – it is not obvious that a user may drag and drop artifacts in the Eclipse environment onto the comprehension trees of the *Goal Explorer* unless they

were trained in its use.

A2: Yes – the user may notice the tooltip while in the *Goal Explorer* that states "... drag java artifacts here ...".

Q: Will users see how to do it?

A1: Yes – if a user is familiar with drag and drop operations in software, they will know how to do so in the *Pollinator* environment.

Q: Will users understand from feedback whether the action was correct or not?

A1: Yes – users will be presented with a dialog to aid in the addition of the artifact as evidence to the comprehension tree and once completed, the evidence will appear under the goal that the user dropped the artifact onto.

A2: No – when users drop unsupported or invalid types of artifacts or elements on the comprehension tree, there is no visible feedback explicitly stating why the element was not added and the user may be misled into thinking something was changed.

- Upon looking inside the class we saw several instance variables of type `Tab` declared that seemed to correspond to the different tab views of Azureus, including the one containing our sought-after context menu, the torrent download screen which was named `mytorrents`. We attached the line of code where this variable was declared to the primary question and went looking for where this instance variable was initialized or created using Eclipse's variable highlighting functionality. Figure 6.2 shows the states of the *Goal Explorer* and source editor up to this point.

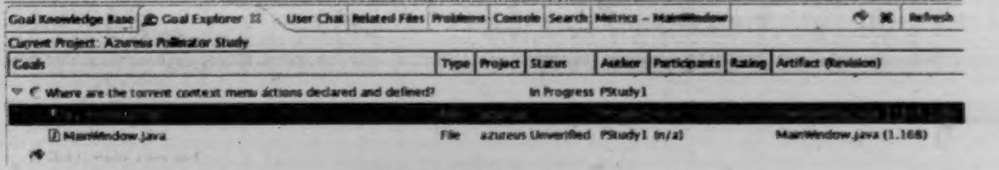
  Q: Will users know what to do?

  A1: Yes – since the user knows that they want to associate this line of code with their mental model, they must "bookmark" it somehow.

  Q: Will users see how to do it?

  A1: Yes – in Eclipse, bookmarks and other visual markers are managed on the left vertical ruler column, so the user may intuitively decide to try opening the context menu on the ruler beside the relevant line of code. They will then see a menu item named "Add Evidence Marker...". Alternatively, the user could open the context menu directly on the line of code and see a menu item named "Attach as Evidence to Goal...". Both actions are functionally identical, but are named according to the context in which they are run.

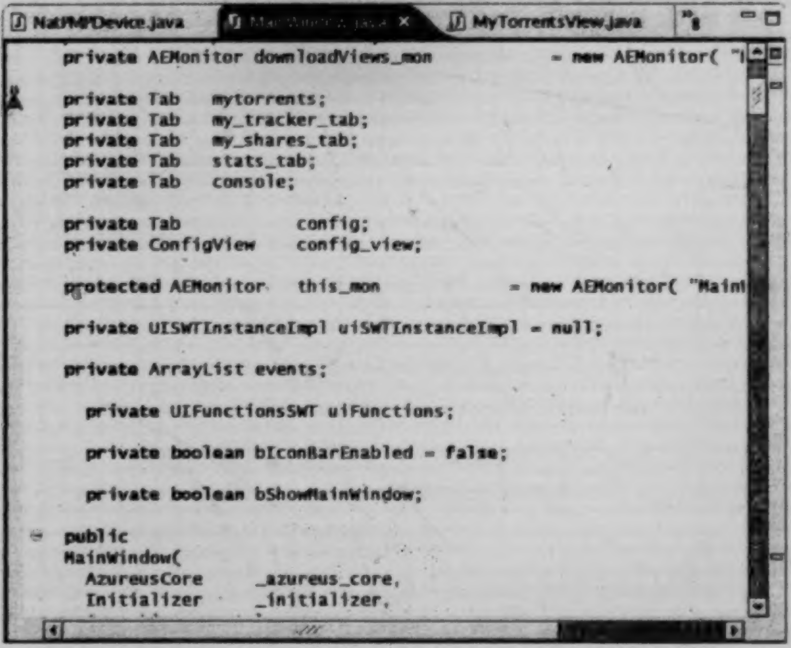
  A2: No – a user may not think of opening a context menu to attach a particular



Current Project: Azureus Pollinator Study

Goals	Type	Project	Status	Author	Participants	Rating	Artifact (Revision)
Where are the torrent context menu actions declared and defined?			In Progress	PStudy1			
MainWindow.java	File	azureus	Unverified	PStudy1	(n/a)		MainWindow.java (1.168)

(a) *Goal Explorer*



```
  private AEMonitor downloadViews_mon           = new AEMonitor( "I
  
  private Tab   mytorrents;
  private Tab   my_tracker_tab;
  private Tab   my_shares_tab;
  private Tab   stats_tab;
  private Tab   console;

  private Tab           config;
  private ConfigView    config_view;

  protected AEMonitor   this_mon           = new AEMonitor( "MainW

  private UISWTInstanceImpl uiSWTInstanceImpl = null;

  private ArrayList events;

    private UIFunctionsSWT uiFunctions;

    private boolean bIconBarEnabled = false;

    private boolean bShowMainWindow;

  public
  MainWindow(
    AzureusCore       _azureus_core,
    Initializer       _initializer,
```

(b) Source editor with line annotation and overview ruler marker.

Figure 6.2: *Goal Explorer* and source editor after adding `mytorrents` evidence

line of code to the comprehension tree.

Q: Will users understand from feedback whether the action was correct or not?

A1: Yes – once the correct action is chosen, the user is presented with a dialog to choose the particular goal they wish to attach the evidence onto and, once completed, the evidence appears in the comprehension tree. If the action was not correct, the evidence would not appear in the desired location.

- About halfway through the `MainWindow.java` file we came to where the `mytorrents` variable was initialized, in a method called `showMyTorrents()`. We discovered that the torrent tab was populated with another view type named `MyTorrentsSuperView`. So, before delving into that class definition, we attached `MyTorrentsSuperView` to the primary question, as a way of leaving a trail or bookmark of where we had consciously chosen to explore as in the other evidence attaching events.

The same usability issues of attaching evidence arise here as they did in the previous actions of attaching evidence, either through drag-and-drop of a Java element or contextual menu action.

- Once inside `MyTorrentsSuperView` we found it was again a composite class made up of two different views of type `MyTorrentsView`.
- After going to the definition of `MyTorrentsView` we attached it as evidence to the primary question. `MyTorrentsView` is a large class of 2992 lines of code, so we used the Eclipse Members view to browse the instance variables and methods searching for something of relevance. After a minute of browsing, we came upon a method named `fillTorrentMenu()` and attached it as evidence.

  The same usability issues of attaching evidence arise here as they did in the previous actions of attaching evidence, either through drag-and-drop of a Java element or contextual menu action.
- Within `fillTorrentMenu()` we found the definition of the context menu actions we were interested in extending. At this point we can say the question has been resolved and change the status of the project's goals and evidence to completed and verified to indicate that they may be useful for others to look at. Figure 6.3 shows the final state of the *Goal Explorer*.

  Q: Will users know what to do?

  A1: Yes – on completing a program comprehension task the user may notice that the status of the goals and evidence in the project need to be updated to a more relevant value to indicate the progress made.

  A2: No – the user may not notice or care to update the progress of the project they are working on since this may not be a usual part of their work routine or is viewed as an annoyance or waste of time.

  Q: Will users see how to do it?

  A1: No – there is no visible means of changing the status without opening the context menu, such as a toolbar button or direct-click action on the status column itself. This may be confusing to users who do not work with contextual menus as the primary method of interaction with data objects.

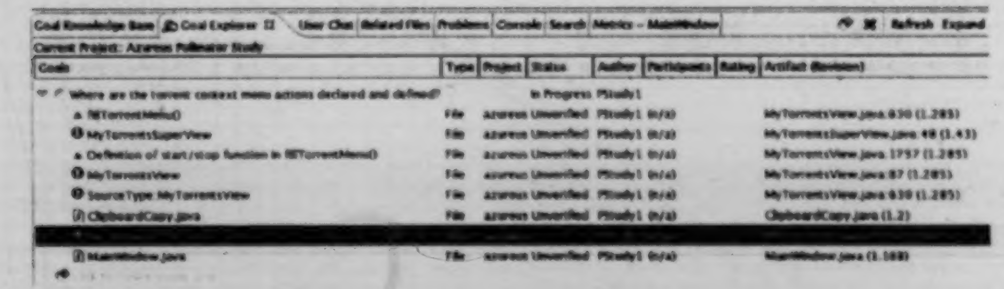


Figure 6.3: The final state of the comprehension tree

A2: Yes – if the user is familiar with using context menus to manipulate data objects such as table items, then it is fairly obvious how to change the status via a context menu named “Mark status”.

Q: Will users understand from feedback whether the action was correct or not?

A1: Yes – the completion of the action gives feedback in two forms. The first is the visible changing of the target object’s status in the *Goal Explorer* view. In the case of changing a status from/to `Invalid/Failed` to/from any other status type, the entire foreground colour of the row changes to further emphasize whether or not the target object is relevant. The second form of feedback is shown within the context menu for changing the object status, through an indication of the current status using a checkmark.

5. With the question answered we could begin the effort to insert code and implement a new action for restarting a torrent download.

The main point to take away from this small walkthrough is that the use of *Pollinator* during a typical software maintenance task did not seem to disrupt or intrude upon the program comprehension process in an overly obtrusive manner, while augmenting the documentation of the comprehension process by preserving it for others and maintaining the connection between a mental model and the software system. By performing a cognitive walkthrough, however, several minor to moderate issues of usability were brought to light that could be addressed to improve the tool.

The cognitive walkthrough covered many scenarios of the tool’s usage except the collaboration features such as user chat, use of the banner view, the knowledge base and the decoration of views with passive awareness information.

More generally, during the course of the task execution we formed impressions of using *Pollinator*, both positive and negative.

Positives:

1. *Integrates nicely with Eclipse's built-in navigation capabilities*

   Code browsing is augmented through the use of Eclipse's ability to find class definitions of the types referenced in the formed question hierarchies. This is not yet integrated with *Pollinator*, so the user currently has to manually type the name of the class or interface.

2. *Subtle usage cues*

   To make the learning curve for using *Pollinator* as small as possible, the *Pollinator* perspective provides a suggested layout for Java development that we found to be most useful. Additionally, there are subtle cues to guide the user such as the extra tree item in the "Goal Explorer" that has a faded colour with the text "Click to create new goal" to make the action of creating goals embedded with the interface of viewing and using them. Also, there are tooltips directly describing usage, such as for the drag-and-drop method to gather evidence in the "Goal Explorer".

Negatives:

1. *Awkward transition from goal hierarchy to local artifacts*

   There should be a simpler way of navigating from the goal hierarchy to local copies of artifacts / evidence. Currently, only remote versions of the files are considered.

2. *Lack of explicit online help or walkthrough of using tool*

   An Eclipse cheatsheet or tutorial could be utilized to walk the user through first-time use of the tool. Contextual help could also be provided.

3. *Consistency in methods for using Pollinator*

   Drag and drop behavior is not an obvious mechanism, even with a tool-tip, however there are multiple ways of doing the same thing (i.e. creating a goal).

Some of the problems with the tool cannot currently be addressed due to limitations in the Eclipse framework. As mentioned earlier, the obtrusiveness of the *Goal Explorer* could perhaps be alleviated by a different perspective layout allowing for more flexibility in sizing views and specifying hiding capabilities. The use of fixed views would solve a problem with the resizing of other views interfering with the size of the banner view, either making it too large or too small.

6.6 Conclusions

Our hypothesis suggests that the comprehension/documenting work context of a developer may be preserved through an explicit knowledge representation that maintains a connection between the mental model of comprehension and the artifacts of the software system. We believe that *Pollinator* preserves work context by making interruptions in thought and of each working session largely irrelevant by integrating a documenting system inline with the developer's normal development environment. Its main advantage over traditional methods seems to be the promotion of even the most basic documenting of program comprehension activities. The knowledge representation format seems feasible in terms of flexibility and understandability. Subsequent evaluation needs to be performed in order to assess other aspects of the tool, such as collaboration, utility, and scalability (performance) as compared empirically to situations with the usage of other similar tools and without the use of any specialized tool at all.

In particular, a thorough test of the collaboration features of the tool could be most fairly accomplished through an experimental scenario that closely approximates the conditions of the observational case study of pair program comprehension. This would also consequently serve as a validation measure of the model of program comprehension gleaned from the analysis of the session data. Such an experiment might be designed such that two subjects would be tasked to work together to understand a software system (e.g. a Perl interpreter). Each subject would be assigned a workstation with the *Pollinator* tool and the Eclipse environment and be situated such that direct face-to-face or voice-to-voice communication between the two is possible. The latter condition better replicates the original case study parameters, but does not fully evaluate whether *Pollinator* is useful when the users are not able to directly communicate with each other except through the tool. Therefore, a separate experiment would need to be conducted with this condition altered accordingly. Ideally, the subjects would also have the same level of computing science background and experience as the participants in the original study. The roles of the subjects could also be pre-designated, so that one subject is assigned to be a director or manager of activities, and the other a technical or domain expert. The director may also serve as the documenter, largely making use of the *Goal Explorer* view while the expert would be more involved in code browsing activities and gathering evidence to be organized by the director.

Without the further evaluation of the collaborative aspects of the tool, our hypothesis cannot be fully verified, but we believe it is possible with additional study and work.

Chapter 7

Secondary Evaluation: Pollinator User Study

7.1 Objective

In order to gain an understanding of how *Pollinator* might be used and evaluated by software engineers, a small user study of the tool was conducted by observing pairs of participants as they completed program comprehension tasks on a subject software system. We hypothesize that the study will show *Pollinator* to be well-received by the participants as an aid to document their comprehension tasks, but may be viewed as obtrusive for easily-completed tasks. The preliminary evaluation in the previous chapter revealed some potential usability problems that the users of this study may also encounter. The study should also provide some comparative insight into how documentation of program comprehension is done with and without a specialized tool for documenting program comprehension.

7.2 Design

A large amount of study material was produced for the conduct of the experiment. See Appendix B for a complete reference on the materials used for the study (i.e. invitation letter, consent form, handbook, task sheets, honorarium form, etc).

Potential study participants were invited to take part in the experiment by invitation (see "Solicitation Letter" in Appendix B.1). The participants of the user study were made up of graduate-level computing science students with experience using Java and the Eclipse development environment.

A questionnaire was conducted before the study to document each participant's general profile of experience with Java and Eclipse. This information might then be used to reconcile any major discrepancies in the observed results of the study. See Appendix B.7 for the questions asked in this questionnaire.

Consent for participation was obtained from participants before beginning their participation in the study. The consent form is attached in Appendix B.2.

The software system used as the subject for the program comprehension tasks is an online voting system written by a fellow graduate student as part of his Master's thesis research into security for electronic voting systems. The system is designed as a web-based application written in Java and JavaServer Pages (JSP) served by a database in the back-end. The JSP files themselves are not shown to the participants to reduce the burden of learning an additional technology. Instead, the Java-translated versions of the JSP files are used.

Three study instances were conducted: a pilot study, a control study, and an experimental study. The pilot and experimental study participants used *Pollinator* while the control study was conducted without a specialized tool for documenting program comprehension.

The pilot study was undertaken to judge the soundness of the experimental design and also improve upon the execution of the study in subsequent instances. The pilot study was an instance of an experimental group that was observed documenting program comprehension using *Pollinator*.

The control study provides a baseline of data for comparison against the experimental study. The participants were given the same program comprehension documentation tasks as in the experimental study, however the participants were not trained or given access to the *Pollinator* tool. The participants were instead asked to complete the tasks using a standard Eclipse Java development environment and to document the task comprehension through written form on the given task sheets.

The objective of the experimental study was to observe how participants use the tool and to gauge the utility and usability of the tool based on feedback obtained at the end of the session.

At the beginning of each instance of the study, participants were oriented on the nature of the study such as its purpose, length, and assurances of rights and anonymity.

In the control study, the participants were given a brief overview of the Eclipse environment, to ensure they were familiar with its basic operations such as file navigation and code browsing. Next, the participants were shown how they might complete a program comprehension task given to them, using a sample system. Once this was complete, the participants were shown an overview of the online voting system. This overview consisted of a listing of the main entities and their interactions, as well as several screenshots of the web interface demonstrating how an administrator or voter would interact with the system. We also briefed the participants on the general nature of the JSP-to-Java translation that the JSP files in the system underwent, so that they would know they would not be looking at any JSP files directly.

The experimental study structure followed the same general sequence as the control study with the addition of a step to introduce the features and use of *Pollinator* as well as changing the demonstration of completing a program comprehension documentation task to use *Pollinator* instead of pen and paper.

Five program comprehension documentation tasks were given for each pair of participants to complete. The program comprehension documentation tasks given for the pilot, control, and experimental sessions are listed in Appendices B.4, B.5, and B.6 respectively. After conducting the pilot session, several revisions were made to the tasks to make the experimental results more useful and reduce redundancy of some of the workload when conducting the control and experimental sessions. Also, some of the task wording was changed to improve the meaning and clarity of the tasks. The only difference in the tasks between the control and experimental groups was the tool for documentation of program comprehension used by the participants.

After the conclusion of the study, the participants' task documentation was compared to reference task documentation produced through careful analysis of the online voting system prior to the start of the study. Our reference analysis was completed with access to additional design documentation provided by the system's author in order to verify correctness.

As mentioned above, several tasks were altered following the pilot session in order to optimize the conduct of subsequent study sessions. In particular, the phrasing of task 1 was altered to include a definition for *business object*. Task 2 was altered to clarify the intent of its objective by specifying the level at which method invocations should be recorded. Task 3 was altered to reduce the number of business objects for which the database access implementation should be documented. It was found that this was a largely repetitive and tedious process through which no additional value was gained in subjecting all of the business objects to this treatment. Task 5 was altered to include more specific direction of the task towards documenting all of the effects of the method `loadAllPosts()` including the internal call to `findPost()`.

Upon completion of the tasks, participants of each study instance were asked to fill out a post-study questionnaire to give their views on the comprehension tasks and the tool or the method of documentation, depending on whether or not the participant was part of the control or an experimental group. See Appendix B.8 for the pilot and experimental group questionnaire and Appendix B.9 for the control group questionnaire.

A handbook for describing the proper conduct of instances of the user study was created to ensure consistency between each instance of the study as well as to improve the preparedness of the experimenters. Appendix B.3 shows the handbook used to conduct the experiments.

7.3 Observations

7.3.1 Participant Profiles

The study participants are anonymously referred to in this thesis by a two character, alphanumeric string. The first character, a letter, designates the study instance the participant took part in, one of "P" for pilot, "C" for control, and "E" for experimental. Within each study instance each participant was given either the number "1" or "2" to differentiate between the participants making up each pair. The second character of the identifier string is this number. The number was assigned randomly, and does not correspond to any identifying or meaningful information concerning the participant.

Here is the information provided by the study participants in the pre-study questionnaire that shows a simple profile of their experience with Java and Eclipse:

Participant Code	**Java Experience**	**Java Knowledge**	**Eclipse Experience**
P1	1 to 2 years	average	occasional use
P2	3 to 4 years	average	occasional use
C1	3 to 4 years	average	occasional use
C2	5+ years	average	everyday use
E1	3 to 4 years	average	occasional use
E2	5+ years	expert	occasional use

Table 7.1: Participant profiles

In summary, all participants in the study had average to expert knowledge of Java and all used Eclipse at least occasionally. This meant that the learning curve for the participants was eliminated in terms of becoming familiar with the basics of either technology. Therefore, the only variable difference between the experimental and control studies in terms of learning experience was the training and use of *Pollinator*.

7.3.2 General Observations

Generally speaking, the participants of the experimental and pilot groups were quick to learn the basics of using *Pollinator* and did not have trouble in understanding and applying the concepts of goals and evidence.

In the pilot study, control of the computer mouse and keyboard alternated naturally between the two participants as they saw fit. Sometimes one person operated the mouse while the other operated the computer. This seemed to coincide with their level of competence and confidence with Java and Eclipse.

This contrasted with the control study where one of the participants was the sole possessor of computer control while the other participant was essentially the documenter. Both participants were still involved in the comprehension process in between documentation actions; however the person without control of the computer was generally restricted to verbal input only.

The experimental study again showed more of the balanced control sharing of the computer that was evidenced in the pilot study. However, when there was sole control by one participant, there seemed to be more direct communication and discussion between the participants on what actions to take next, guided by the structure of documenting comprehension using *Pollinator*.

In the case of the study instances where *Pollinator* was used, the participants took virtually no written notes, even though paper and pen were available for this purpose. Instead, the participants were able to use *Pollinator* for both drafting up their answers, by framing their comprehension in terms of goals and evidence, and firming up their answers after further investigation. In the control study, the participant who was not in control of the computer took notes in parallel with the actions done on the computer, however often the person on the computer waited for the note taker to finish writing before being able to proceed. The written notes were in addition to the final answers written down on the task sheets.

7.3.3 Study-specific Observations

In this section, an anecdotal summary of interesting events and patterns that occurred in the sessions through the course of completing the comprehension tasks is documented . The same participant code identifiers shown in Table 7.1 are used here.

Pilot

After observing the interaction and actions of the participants, it became apparent that P1 was more of an expert on the use of Eclipse, while P2 was more of an expert in the domain of databases. As a result, P1 was more involved in control of the computer due to his familiarity with Eclipse, while P2 took a more directorial role in terms of framing the higher-level context and concepts.

Some other general observations of the participants was their double-checking of each other's work to ensure that they both had the same understanding of a concept or fact. Once given a task, the participants talked with each other about the meaning of the tasks given and discussed which steps to take next in the documentation process.

No hand-written or electronic notes of any kind were made by the participants, even though paper and pen was available for use. Documentation was completed exclusively through *Pollinator*.

In conducting the pilot study, we were able to observe behavior and receive feedback that allowed us to alter the program comprehension tasks and the environment of the study to ensure a better experience for the participants and a more focused design to avoid redundant observations.

In particular, we learned that the third task, which was about documenting where the database access was implemented for the business objects of the system, was very repetitive and tedious for the participants because the procedure of looking up and documenting the implementation was the same for each business object. From an experimental point of view, this repetitiousness did not provide any further insight into the evaluation of the tool and thus this task was changed to limit the number of business objects documented to three from a maximum of six.

We also saw that one of the tasks was too vague in its wording and we did not get the desired answer from the participants. The sixth task asked for the effects and post conditions of the method `loadAllPosts()`, however it was assumed that the participants would also document the behavior of methods called from `loadAllPosts()`, in particular the method `findPost()`. This additional documenting need makes the task far less trivial and gives us more information to work with, therefore the task was changed to specifically state that `findPost()` should also be documented.

As for the tool itself, we learned that a large usability problem with *Pollinator* was the inability to edit previously created text evidence. This was deemed such an obvious and severe bug as to impede the participant's useful interaction with the tool, that it was fixed for the version of *Pollinator*used in the subsequent experimental study.

Upon completion of the pilot study, it became apparent that an easy way to look at the comprehension documentation produced by different groups was needed. It looked to be cumbersome to switch between different databases when wanting to compare the different studies. Therefore, we decided to make an export function for the Goal Explorer so that its contents (the documented comprehension tree) could be viewed outside of the tool and in a plain text file.

Control

The control group, and especially the participant C2 took a far more top-down or high-level approach to program comprehension than the other groups. For example, at the beginning of the first task, C2 states that business objects can generally be found in the middle layer of a three-tier system. The participants then proceeded to undertake a general overview of the system's architecture before looking more closely at the files and implementations. This group's top-down approach was also evident in the way in which they completed the third task. Instead of going through the business class objects linearly looking through the code for database implementation, this group decided to instead go directly to the resource bundle file where the SQL queries were defined (`SQLQueries.properties`) and use Eclipse's search feature to lookup the implementation of each query. Therefore, the control group ended up making much more extensive use of Eclipse's search functionality, but took much longer to complete the third task than the other two groups. This same technique was used in the fourth task, to look up the location for the code implementing authorization for administrator access.

C2 was also interested in the higher-level aspects of the system, which was seen in his asking for the business rule for the procedure of registering a new voter.

Some issues arose in this study instance whereby the participants were confused by a task's wording or concept used in the task. For example, in the first task, C2 asked what the imperative "document" meant in the context of the task. He felt it was somewhat vague and did not specifically say what needed to be written down. Also in the first task, the participants did not understand the intent of the term *business object*, taking it to mean that the objects should only be domain-specific in nature. This led them to exclude the `Person` class from their answer, which was incorrect compared to the reference answer.

Experimental

Before starting the tasks, E2 gave a suggestion that there should be a note-taking area or component of *Pollinator*. This suggestion was made while the experimenter was giving an overview of the e-voting system. E2 took some notes on the paper provided during this process, suggesting he would have used a computer-based tool to do so if it were available.

In the experimental group, we observed E1 to control the computer most, if not all, of the time.

Compared to the pilot study, the participants were much more critical of the code and design of the subject system. While browsing through the code, the participants sometimes made comments on the quality of the system's architecture and design. These comments were not included in the documentation produced by the participants.

On observation of the participants' use of the tool, we noted that a better visual cue is needed when there is no project open, to suggest to the user that they need to open the project explorer. The participants needed assistance in this step, as they assumed that a *Pollinator* project would be created from the Eclipse workspace itself, like any other Eclipse project. This potential problem was also mentioned in the cognitive walkthrough done in Section 6.5

Several suggestions to improve *Pollinator* arose out of this study instance. E1 suggested that *Pollinator* should provide the full Java package "path" to named evidence entities when

	Task 1 /6	**Task 2 /6**	**Task 3 /3**	**Task 4 /2**	**Task 5 /2**	**Total /19**
Pilot	6	6	3	2	2[a]	19
Control	5	6	2	2	2	17
Experimental	6	5	3	2	2	18

---

[a] This task was more vague in the pilot study and did not specify to document the effects of `findPost()`

Table 7.2: Task scores by group

adding them in order to easily distinguish these entities from others with the same common name. For example, there are multiple `_jspService` methods through all the `.jsp` files, so *Pollinator* should provide the full path to these instances when adding them as evidence.

After the study, E2 suggested that a pre-defined question set would be beneficial, to reduce the burden of creating goals. For example, a common question for any program understanding project or task might be "What are the business objects?". E1 also suggested the addition of domain-specific questions, if possible.

7.4 Analysis

In this section, we analyze the data obtained from the studies to gain an understanding of how effective *Pollinator* is in aiding program comprehension documenting by comparing the observations and results of the control to the experimental study. This comparison will look at the correctness of documentation produced, the effort expended on each task and overall, and the thoughts of the participants themselves as conveyed through the completion of the post-study questionnaire.

7.4.1 Task Analysis

Appendix C contains the details of a descriptive comparison of the tactics and methods employed by the pairs in the course of completing the given program comprehension tasks. Each task is looked at individually, with the documented comprehension produced by each pair shown along with the correct answer.

Summary of task analysis

The documentation produced by the three groups for each task was scored and the total tabulated in table 7.2. There is very little variation on the scores obtained by each of the groups. In fact all groups are within two points of each other within a total of 19. It should be noted, though, that the control group fared the poorest by failing to document one of the business objects for task 1 and one of the database implementation locations for task

Group	**Task 1**	**Task 2**	**Task 3**	**Task 4**	**Task 5**	**Total**
Pilot	6:05	10:45	22:21	3:57	8:02	**51:10**
Control	13:45	15:35	20:00	3:50	5:30	**59:40**
Experimental	13:05	13:10	15:10	7:45	23:40	**1:13:00**[a]

---

[a] The experimental group exceeded the allotted time of one hour for task completion. See threats to validity below.

Table 7.3: Time spent for each task

3. These errors may be attributed to the misinterpretation of what was intended by the term "business object" and the methodology the control group used to look up the database implementations.

Table 7.3 shows the amount of time spent per task by each study group. The data suggests that groups working with *Pollinator* fared better than the control group on tasks 1, 2, and 3. The control group, on the other hand, completed tasks 4 and 5 in less time than both the pilot and experimental groups.

An analysis of the nature of the different tasks may give insight into the discrepancies in time allocation. Firstly, tasks 1 and 3 require the participants to browse through almost all of the code base in order to complete the documentation. This contrasts with tasks 4 and 5 which involve very localized pieces of code that are contained in one file. Task 2 falls somewhere in between, as it specifies a particular file to look at but the method calls being documented are implemented elsewhere in the system. This may indicate that *Pollinator* is helpful in tasks where the mental context is large and broadly-based throughout a software system. The on-screen presence of the task in a structured, hierarchical form may reduce the occurrences of context losses and the need to reorient. However, it appears that when the subject matter of a task is very focused on a particular aspect or file of a software system, *Pollinator* may actually become a hindrance to the user. For example, the time it takes to form a task into a goal or explicit question may instead be used to actually search out the answer the question. This related "overhead" then becomes much more noticeable in tasks that take relatively little time to complete.

There are some data points in table 7.3 that bear some explanation as they appear to be outliers. For instance, the pilot group seems to take an extraordinary amount of time to complete task 3, even compared to the control group. However, as mentioned earlier, task 3 in its original pilot form asked for *all* six business objects to be documented for database access instead of the three requested in the control and experimental groups. Also of note is the experimental group's large time to complete task 5. This may be due to the fact that the

	P1	P2	C1	C2	E1	E2
Task Difficulty	average	easy	average	average	easy	easy
Pollinator Usability	very good	good	n/a	n/a	good	good

Table 7.4: Participant feedback

experimental pair seemed to read too much into the question and spent a significant amount of time examining and discussing the code when it appeared to the experimenter that they already had enough information early on in the task to complete it.

7.4.2 Participant Feedback

Based on questionnaires given to the study participants at the conclusion of each session, we received valuable feedback regarding *Pollinator*, the conduct of the experiment, and suggestions for how documenting program comprehension could be improved through the use of a tool. The participants' detailed responses to the questionnaire are contained in Appendix D.

Questionnaire Summary

Pollinator seemed to be useful as an aid by alleviating the overall perceived difficulty of the program comprehension tasks. Based on the feedback received in the post-study questionnaire, both participants in the control group rated the task difficulty as "average", while three of the four participants in the pilot and experimental groups working with *Pollinator* regarded the tasks as "easy" and the remaining experimental participant rated the tasks as "average". This suggests that *Pollinator* is at least better than having no tool in making the tasks seem easier to complete.

Several reasons were given as to why *Pollinator* was helpful in documenting program comprehension, mostly centered around its structured goal and evidence framework. The participants thought that creating goals and adding evidence made the process more organized, with one participant describing being able to attack the task in a "divide-and-conquer" fashion with the tool. Another participant saw the potential for *Pollinator* to be used on a large project where many programmers or maintainers are employed, if used in a consistent manner.

The participants also found *Pollinator* to be fairly usable with three of the four participants rating the tool's usability as "good" and one giving the tool a rating of "very good". Criticisms about its usability included its lack of ability to directly open the workspace source code from *Goal Explorer*, the difficulty of switching between the perspectives and views, and the lack of multiple-select actions such as marking more than one piece of evidence as verified at once.

All of the participants found *Pollinator* to be helpful in coordinating communication between themselves and their partner in the study. Three of the comments noted that the goals and evidence provided a common platform on which to base discussions upon for the task at hand. This reduces incidences in which the participant's cognitive models for comprehension conflicts with each other, allowing for more time spent on discussing the real issue at hand rather than forming mental context.

Participants liked *Pollinator*'s tight integration with Eclipse, particularly the ability to drag-and-drop entities between Eclipse and the *Pollinator* views. Allowing users to drag code elements contributed greatly to the perceived ease of use. The participants again emphasized their appreciation for *Pollinator*'s structured knowledge representation as a strength in allowing mental context and knowledge to be quickly assimilated in the future as necessary.

When asked if they would use *Pollinator* in their own work and/or research, participants were very receptive. One suggested he would use it on a big multi-developer project, while another said that *Pollinator* would be useful in his research that uses a public parser, helping to reduce the effort needed in program comprehension when modifications to the parser are made. One of the participants would use *Pollinator* if it was released as open source.

Three of the participants submitted additional feedback and suggestions beyond the specified questions of the questionnaire. Two higher-level suggestions were made concerning *Pollinator*'s applicability to more general projects and research. One was that *Pollinator* should be language-independent or support different languages. The other was that it should be more formalized, allowing for formal reasoning into features through more constrained pieces of evidence. The other suggestions were in regards to *Pollinator*'s interface and feature set. One of the participants thought that the goal creation dialog should be made consistent with the *Goal Explorer* view in terms of the hierarchical display of the goals. Another desired the ability to attach more complex evidence explanations that contain pictures or UML diagrams.

The control group feedback allows us to compare their impressions of the program comprehension tasks with those of the experimental groups and make some judgments on the influence of *Pollinator* on those impressions. It was already mentioned that the control group generally found the tasks to be slightly more difficult than the experimental group. The control group was split in their opinion on whether or not this "manual" approach was useful and reliable. The dissenter noted that the method may be unreliable because the underlying business logic of the system is not well understood, but this is a common issue for both groups.

As for the communication between the partners, again the control group was split as to how their method of program comprehension affected the coordination of discussion. The partner writing down the documentation thought the communication was quite good, while the other complained of having to wait for the other partner to finish writing.

Both control group participants thought that a specialized tool could help in completing the tasks. The partner writing down the documentation thought that a tool containing the logic behind the system design would help, while the non-writer believed that better tool support for documentation would be nice. Taken together with the previous feedback on partner communication, it would seem that the person who is not writing down the information sees more benefit in a tool that supports alternative documentation methods than the writing partner.

Both also thought that a specialized tool could help coordinate communication between the partners, with one of the participants qualifying his statement by saying that it would only be useful if the people were not on the same computer. The other thought that a tool could help in aligning the teammate's ideas of the comprehension with each other, rectifying disagreements in opinions.

7.5 Threats to Validity

Several threats to the validity of this study exist. Perhaps most prominently noticeable is that the study is based on a relatively small sample size. With one control group and one experimental group there are essentially only two data points for comparison. The pilot group study does add some data available for analysis, however because of some of the differences in how it was conducted, it cannot be directly used to compare against the control group.

Another problem arises, in that although the participants come from very similar backgrounds, they may have different experience and knowledge that is not quantifiable. For instance, in the control group, it appeared that one of the participants may have had more experience with the architecture of web-based applications written using JSP and Java.

"Unfresh" study participants that have seen previous presentations or other material on the research in this thesis may be biased in their level of knowledge of the tool and program comprehension research. However, all participants are given the same training on the tool, and the practical use of the tool was not illustrated in the group presentations.

Another source of uncertainty is the possibility of inadvertent bias and hinting exhibited by the experimenter during the study sessions. For example, in the training portion of each session there may be slight variations from instance to instance that either gives a bit more or a bit less information to the participants on either how to use the tool, how to answer the questions, or other knowledge. This problem was mitigated by the use of the handbook to try to ensure consistency between the different groups through guidelines and procedures. However, as an explicit script was not used this means that the wording of the training differs between groups, possibly leading to inconsistency.

As mentioned in the footnote to table 7.3, the experimental group technically exceeded the one hour allotment that was given in the handbook. However, as explained, we believe that the pair had over complicated the task and was likely prepared to complete it well before the time expired. Additionally, tasks 1 to 4 were completed well under an hour with just over 10 minutes remaining to complete task 5. We do not believe this data point to compromise the comparison between control and experimental groups, however it is a possible threat to validity in that had the time been more strictly adhered to, the experimental group may not have completed task 5 correctly. In this case, the experimental group would have scored the same or one point lower than the control group in terms of correctness.

7.6 Study Conclusions

This study was conducted with the objective of evaluating the utility and usability of *Pollinator* as a helpful tool in documenting program comprehension.

From our observations of the use of *Pollinator*, we have seen that it seems a tool best suited for large tasks, that encompass a lot of code in a software system. In small, short tasks that involve only one or two files, the overhead of using *Pollinator* may outweigh its benefits in terms of task completion time and efficiency. However, based on our analysis of the quality and correctness of the documentation produced for the tasks we believe that *Pollinator* holds a slight edge over the control group method using no specialized tool. This edge is very slight indeed and should not be viewed as an authoritative differentiator.

Participants of the study found *Pollinator* to be usable and useful, with some minor criticisms of the interface and features. Comparing the ratings of task difficulty between the control and experimental groups we saw that the users of *Pollinator* thought the tasks were easier than those of the control group.

In general terms, the participants using *Pollinator* thought that it was very usable, ranking it good to very good. They also liked its utility, especially with regards to the goal and evidence structure that they felt helped organize their thoughts and documented knowledge.

Several lessons were learned conducting this user study, that could be applied to future instances.

Through the pilot study, we learned that the interpretation of a written question or task may be completely different from the original author's intent. Subtle variations in the words used and ideas emphasized can have a large impact on the meaning conveyed.

We also learned through the pilot study that tedious and repetitive tasks can lead to the study participants becoming unfocused and bored which may affect their ability to complete the current and future tasks. It seems best to keep the participant's attention through variety and reasonably sized tasks.

Based on our user study, we believe that *Pollinator* is already quite a useful tool, but can be improved and expanded in order to become more generalized and useful to researchers and engineers alike. Some of the future work comes directly from the user feedback, such as covering all or additional programming languages, formalizing the evidence structure, allowing more types of evidence content, and improving the interface to a more polished level.

In terms of validating our abstract model of the phases of comprehension from Chapter 4, we believe that our observations of the participants follow the model entities quite closely. In particular, we noticed that the cycle of goal generation including implicit and explicit questions was represented in the data of the *Pollinator* study. The participants generally took the task as their primary goal and then proceeded from there to either look directly at the code (implicit question) or generate new questions and goals (explicit question) to guide further investigation.

Chapter 8

Conclusions and Future Work

In this thesis we presented work on research into supporting the documentation of program comprehension in collaborative settings, with the goal of addressing communications and knowledge persistence problems in software maintenance.

We presented the observations and analysis of an observational case study on pair program comprehension. The resulting observed scenarios, event magnitudes, and event relationships were used to motivate the development of a comprehension structure and also the requirements for a cognitive support tool.

A prototype collaborative cognitive support tool, Pollinator, has been implemented as an Eclipse plug-in for Java development and partially evaluated. The preliminary evaluation of the tool showed that it met many requirements for a collaborative program comprehension documentation tool. Compared to other similar tools, only Pollinator seems to currently have the capabilities that merge the needs of both cognitive support and collaboration. Heuristic and cognitive walkthrough evaluations assessed the usability of the tool and noted some deficiencies, while the walkthrough also demonstrated the application of Pollinator to a real-world system. The usability evaluations seem to show that the use of Pollinator does not grossly intrude upon the normal process of software maintenance and was somewhat useful as a mechanism in documenting the comprehension task progress.

The *Pollinator* user study allowed us to incorporate another degree of evaluation into the analysis of its feasibility as a collaborative support tool for documenting program comprehension. Observation and direct user feedback show that *Pollinator* is a useful and usable tool, especially in regards to assisting collaboration between partners, reducing barriers to building and maintaining mental contexts, and working on large, complex tasks.

One of *Pollinator*'s major shortcomings may be its seeming hindrance as an aid in regards to small, focused program comprehension tasks. However, we believe that the long term benefits of having persistent documentation may outweigh this initial investment in the overhead of using a specialized tool.

Future work includes considering how social tagging [39] and keyword databases may be used to assist in structuring the knowledge base. Incorporating community or team-defined tags would augment the collaborative support provided, by allowing multi-dimensional categorization of knowledge that is closely associated with the team's own understanding of the domain and code. The current search capabilities of the tool are limited, and it may be useful to have a recommendation system be a part of the tool to discover existing relevant goals. Part of the recommendation system could be the automatic gathering of evidence or knowledge such as in other tools described in this thesis. More advanced search functionality could include automatically linking questions or goals that are related, so that effort is reduced when a question or goal has already been resolved.

Implementing generalized programming language support into *Pollinator* is an idea that is considered for future work and was also mentioned in the feedback of the user study. An alternative to this would be the selective extension of *Pollinator* to support specific languages such as C++ and C#.

Bibliography

[1] Tom Allen. *Managing the Flow of Technology.* MIT Press, 1977.

[2] Grady Booch and Alan W. Brown. Collaborative Development Environments. Rational Software Corporation, October 2002.

[3] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies*, 18(6):542–554, 1983.

[4] Li-Te Cheng, Susanne Hupfer, Steven Ross, and John Patterson. Jazzing up eclipse with collaborative tools. In *OOPSLA Workshop on Eclipse Technology eXchange*, pages 45–49, 2003.

[5] Michael J. Coblenz, Andrew J. Ko, and Brad A. Myers. Jasper: An eclipse plug-in to facilitate software maintenance tasks. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange (eTX 2006)*, 2006.

[6] Carl Cook. Collaborative software engineering: An annotated bibliography. Technical report, University of Canterbury, June 2004.

[7] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[8] Victor M. González and Gloria Mark. Constant, constant, multi-tasking craziness: managing multiple working spheres. In *CHI '04: Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, pages 113–120. ACM Press, 2004.

[9] John C. Henderson and Jay G. Cooprider. Dimensions of I/S Planning and Design Aids: A Functional Model of CASE Technology. *Information Systems Research*, 1(3):227–254, 1990.

[10] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. An empirical study of global software development: Distance and speed. In *23rd International Conference on Software Engineering*, page 0081, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[11] Ric Holt. Software architecture as a shared mental model. In *Proceedings of the ASERC Workhop on Software Architecture*, University of Alberta, August 2002.

[12] Singer J., Elves R., and Storey M.-A. NavTracks: Supporting Navigation in Software Maintenance. In *ICSM'05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334. IEEE CS Press, September 2005.

[13] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM Press.

[14] Jürgen Koenemann and Scott P. Robertson. Expert problem solving strategies for program comprehension. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 125–130. ACM Press, 1991.

[15] S. Letovsky. Cognitive Processes in Program Comprehension. *Empirical Studies of Programmers*, pages 58–79, 1986.

[16] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Papers presented at the first workshop on Empirical Studies of Programmers*, pages 80–98. Ablex Publishing Corp., 1986.

[17] Robert Lougher and Tom Rodden. Supporting long-term collaboration in software maintenance. In *COCS '93: Proceedings of the conference on Organizational computing systems*, pages 228–238, New York, NY, USA, 1993. ACM Press.

[18] Jun Ma. Building reverse engineering tools using Lotus Notes. Master's thesis, University of Victoria, 2004.

[19] Jun Ma, Holger M. Kienle, Piotr Kaminski, Anke Weber, and Marin Litoiu. Customizing lotus notes to build software engineering tools. In *CASCON '03: Proceedings of the 2003 Conference of the Centre for Advanced Studies*, pages 211–222. IBM Press, 2003.

[20] Michael Marcotty. *Software implementation*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1991.

[21] A. H. Maslow. A theory of human motivation. *Psychological Review*, 50:370–396, 1943.

[22] Chapin N. Software maintenance: a different view. In *AFIPS Conf. Proc. 54 National Computer Conference*, pages 509–513, 1985.

[23] Jakob Nielsen. How to conduct a heuristic evaluation. `http://www.useit.com/papers/heuristic/heuristic_evaluation.html`.

[24] Jakob Nielsen. Ten usability heuristics. `http://www.useit.com/papers/heuristic/heuristic_list.html`, 2005.

[25] Jakob Nielsen and Robert L. Mack. *Usability Inspection Methods*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[26] C. Parnin and C. Görg. Building Contexts From Interaction History for Recovery and Exploration during Program Comprehension. In *Proc. of 14th IEEE International Conference on Program Comprehension (ICPC)*, Athens, Greece, 2006.

[27] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cog. Psychol.*, 19:295–341, 1987.

[28] Jennifer Preece, Yvonne Rogers, and Helen Sharp. *Interaction Design: beyond human-computer interaction*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[29] Eclipse Communication Framework Project. `http://www.eclipse.org/ecf`.

[30] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *10th International Workshop on Program Comprehension (IWPC 2002)*, pages 271–278. IEEE CS Press, 2002.

[31] Roger M. Ripley, Ryan Y. Yasui, Anita Sarma, and Andr&#233; van der Hoek. Workspace awareness in application development. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 17–21, New York, NY, USA, 2004. ACM Press.

[32] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416. ACM Press, 2002.

[33] Martin P. Robillard and Frédéric Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In *Proceedings of the Eclipse Technology Exchange at OOPSLA*, October 2005.

[34] Pierre N. Robillard, Patrick d'Astous, Françoise Détienne, and Willemien Visser. Measuring Cognitive Activities in Software Engineering. In *1998 (20th) International Conference on Software Engineering*, April 1998.

[35] Sangam. `http://sourceforge.net/projects/sangam`.

[36] Anita Sarma. A survey of collaborative tools in software development. Technical report, University of California, Irvine, March 2005.

[37] Anita Sarma, Andre van der Hoek, and Li-Te Cheng. A need-based collaboration classification framework. In *1st Workshop on Eclipse as a Vehicle for CSCW Research*, 2004.

[38] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, pages 595–609, September 1984.

[39] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. Waypointing and social tagging to support program navigation. In *CHI '06: Extended Abstracts on Human Factors In Computing Systems*, pages 1367–1372. ACM Press, 2006.

[40] Margaret-Anne D. Storey. Theories, methods and tools in program comprehension: past, present and future. In *13th International Workshop on Program Comprehension (IWPC 2005)*, pages 181–191. IEEE CS Press, 2005.

[41] Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418. IEEE CS Press, 2003.

[42] Arie van Deursen. Program comprehension risks and opportunities in extreme programming. In *Eighth Working Conference on Reverse Engineering*, pages 176–185, 2001.

[43] Iris Vessey and Ajay Paul Sravanapudi. Case tools as collaborative support technologies. *Commun. ACM*, 38(1):83–95, 1995.

[44] A. von Mayrhauser and A. M. Vans. Industrial experience with an integrated code comprehension model. *Software Engineering Journal*, 10:171–182, 1995.

[45] Andrew Walenstein. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. *11th International Workshop on Program Comprehension (IWPC 2003)*, page 185, 2003.

[46] Laurie Williams and Robert Kessler. *Pair Programming Illuminated*. Addison Wesley, 2002.

Appendix A

Implementation Details

A.1 Pollinator: Extensions

This section lists the extension points used by the Pollinator plug-in to implement the tool's functionality.

- `org.eclipse.ui.views`

  Extended to implement six views (*Goal Explorer*, *Goal Knowledge Base*, *Pollinator Awareness*, *Pollinator Session History*, *User Chat*, *Related Files*) and one category (*Pollinator*).

- `org.eclipse.ui.perspectiveExtension`

  Extended to declare and define the *Pollinator* perspective layout.

- `org.eclipse.core.runtime.preferences`

- `org.eclipse.ui.preferencePages`

  Extended to provide some basic mechanism for persisting and changing login information and preferences.

- `org.eclipse.ui.actionSets`

- `org.eclipse.ui.bindings`

- `org.eclipse.ui.commands`

- `org.eclipse.ui.handlers`

  Extended to declare global actions, such as a keystroke-shortcut for creating a new question in *Goal Explorer*.

- `org.eclipse.ui.popupMenus`

  Extended to declare context menus and connect them to actions in various contexts, such as the editor's ruler and text area and whenever a popup menu is displayed on a resource.

- `org.eclipse.ui.editors.annotationTypes`

- `org.eclipse.ui.editors.markerAnnotationSpecification`

  Extended to declare a new annotation type used to denote evidence in the source code and other artifacts.

- `org.eclipse.ui.editorActions`

  Extended to connect the selection event on an annotation that is clicked to the appropriate action that loads the context of the evidence in the *Goal Explorer*.

- `org.eclipse.ui.decorators`

  Extended to define decorators to be placed on resources that are open by users of the Pollinator system in the appropriate views (i.e. *Package Explorer*, *Navigator*, and *Members*).

Appendix B

User Study Materials

B.1 Solicitation Letter

Dear xxxx (fill in name of potential participant),

You are being invited to participate in a study entitled "User Study of a Collaborative Program Comprehension Tool". The purpose of this study is to evaluate the usability and effectiveness of a program comprehension tool that we developed to aid in the documentation of program comprehension in collaborative development environments.

We are interested in your participation because you have experience working with Java.

The study will consist of several steps. At the beginning of this study, we will introduce background knowledge related to our tool, such as the design and manipulation of the program comprehension documentation structure and the use of embedded hooks into our tool from various parts of the Eclipse environment. We will provide examples of how the tool could be used in some sample scenarios. Once you have been given an opportunity to try the tool yourself, we will present a set of questions about the software system that we will use for the study and ask you and your partner to answer them using the provided tool and desktop environment. Answer the questions as best you can. Afterwards, you will be presented with a brief questionnaire so that we may obtain your impressions of the tool. Once this is complete, there will be a short debriefing where you may bring up any questions or concerns you have directly to us. We anticipate that this whole process will take about 90 minutes to 2 hours (at most) to complete. The proceedings will be videotaped in order to observe the type of activities you engage in during the course of using the tool. The resulting video footage will only be used for this research project and only accessible by the researcher (Benjamin Chu) and his supervisor (Kenny Wong). The video will be destroyed after review by these individuals and will not be presented or viewed by anyone else. Your participation in the study is completely voluntary. You can withdraw from the study at any time, without explanation. You can take breaks and ask questions at any time during this study. Any data collected in the study will remain confidential and be kept on secure machines. Any analysis results that would be published will remain anonymous (that is, not identifiable to you).

If you are interested in participating in this study or have further questions, please contact me by email: bechu@cs.ualberta.ca or by phone (780) 893-6586.

Thank you and I look forward to hearing from you.

Regards,

Ben Chu

M.Sc. Student

Department of Computing Science

University of Alberta

B.2 Consent Form

CONSENT FORM FOR PARTICIPATION IN STUDY OF PROGRAM COMPREHENSION TOOL

We invite you to participate in a small research study on collaborative program comprehension. The purpose of this study is to contribute to the evaluation of a software tool ("POLLINATOR") created by the researcher. POLLINATOR is meant to help in the documentation and dissemination of acquired program comprehension knowledge in a software engineering environment.

This study is being undertaken as part of Benjamin Chu's research towards completion of a Master of Science degree. Dr. Kenny Wong is the supervising researcher on this project.

METHODOLOGY

The method of the study will consist of observing a pair of research participants as they complete given program comprehension task(s) of a subject software system. The tasks will consist of questions posed to the participants concerning various aspects of the design and implementation of the software system.

Prior to the initiation of program comprehension task activities, you will be given a brief overview of the use of POLLINATOR so that you may become familiar enough with the tool to use it to complete the tasks.

As a pair, you will be given access to a single computer with 2 LCD monitors. The setup you will be presented with will consist of a typical Windows or Linux OS desktop environment and the integrated development environment, Eclipse, in which POLLINATOR is implemented. You will then be asked to complete the task(s) using POLLINATOR within this environment on the target software system.

The session will be video-taped with a camcorder mounted on a tripod that is situated behind you so that we may record your activities and also zoom in on the computer screen as necessary to observe your interactions with POLLINATOR and the computer.

Once you have answered the questions as best you can, you will be debriefed and asked to fill out a brief (10 question) post-session questionnaire about your experiences during the session.

We anticipate the entire process to take about 90 minutes to two hours (at most).

RIGHTS

You have the option to decline participation in the study for any reason and to withdraw at any time without prejudice to pre-existing entitlements, if any, and to continuing and meaningful opportunities for deciding whether or not to continue to participate.

This study is independent of any course or seminar you may be enrolled in and your acceptance or withdrawal of participation will not be disclosed to anyone other than the primary researcher.

If the option to decline is exercised, any collected data will be withdrawn from analysis and not included in the study. The data will then be subsequently destroyed.

You have the right to anonymity and confidentiality throughout the course of the research study. Proper safeguards for securing data will be taken and any identifying data will be destroyed once it has been properly anonymized. Prior to completion of the study, only one (1) copy of the video will be stored on DVD format kept inaccessible to anyone but the researcher. One (1) copy of the answer data you provide through the completion of the comprehension tasks will be kept securely with the researcher until the data is anonymized, at which point it will be irreversibly destroyed. After the completion of the study, the anonymously coded data will be retained for a period of one (1) year or until the research analysis is complete, whichever comes first.

Any recorded dialogue used in the content analysis from the session will be of a general nature that would preclude participant identification. In order to ensure your anonymity and privacy, we ask that you do not discuss the activities and interactions between you and your participant partner with any other individuals.

The video obtained through this study will only be used for the purposes of this research project and only accessible by the researcher (Benjamin Chu) and his supervisor (Kenny Wong). The video will be destroyed after review by these individuals and will not be presented or seen by anyone else.

You also have the right to the results of the study at any point after its completion.

USAGE

The data obtained in this study is intended to be used for the purposes of software engineering research at the University of Alberta.

CONTACTS

If you need to contact someone with regards to concerns, complaints or other consequences of the study you may contact any and all of the following individuals:

Benjamin Chu
Primary Researcher

MSc. Student
(780) 492-6909
bechu@cs.ualberta.ca

Dr. Kenny Wong
Supervisor
Associate Professor
(780) 492-5202
kenw@cs.ualberta.ca

Dr. José Nelson Amaral
Professor - Associate Chair (Graduate Studies)
(780) 492-5411
amaral@cs.ualberta.ca

SIGNATURE

By signing this form, you are agreeing to participate in this research study within the rights and restrictions outlined above.

Name: ____

Signature: ____________________ Date: ____________________

Thank you for your participation.

B.3 User Study Handbook

B.3.1 Content/Overview

This handbook describes the content and guidelines for conduct of the *Pollinator User Study* being conducted as part of the research for Benjamin Chu's Master of Science thesis.

The purpose of the study is to evaluate the collaborative program comprehension research tool *Pollinator* on aspects of usability, utility, and collaboration.

The evaluation will be determined through an analysis of the execution of tasks by user participants on a given software system.

The baseline or control group will serve as a reference point, as this pair of participants will be directed to complete the tasks without the use of *Pollinator*, using a plain Eclipse development environment. This group consists of one pair of participants.

Each experimental group will consist of a pair of participants directed to complete the program comprehension tasks by using *Pollinator* within the Eclipse development environment.

An experimental *session* is defined to be a single instance of the execution of this study with either a control group or an experimental group.

Each session is designated to last about 2 hours in length. The first hour of the session is assigned to introduction and orientation of the study content and expectations presented to the participant pair. In the case of the control group, this segment may be significantly shorter than that needed for experimental groups due to the exclusion of training time needed for the use of *Pollinator*.

The second portion of the session, designated to one hour, will be devoted to the participants' execution of the given program comprehension tasks.

Finally, when the tasks have been completed or when one hour is spent, the participants will be asked to complete a short post-study questionnaire regarding their experiences and opinions on using *Pollinator*. The participants will then be given a 5 minute debriefing and thanks for participating in the study and given a small honorarium, signing a sheet for acknowledgement of receipt of the honorarium.

Total time allocation is expected to be no more than 2 hours.

B.3.2 Study Protocol

Each session will be fully video-taped using a digital camcorder mounted on a tripod behind the participants and their workstation. The position will be such that both participants and their actions are in frame. Additionally, the contents of the computer monitors will be filmed at opportune moments using the zoom feature, such that major UI elements (text, buttons, cursor) are discernible.

Orientation

As an introduction to the experiment, state the following about its objectives and structure:

- The purpose of the study is to gather data on pair program comprehension, specifically as a measure of the usability, utility, and collaborative capabilities of the research tool, *Pollinator*.
- The entire session will last about two hours, but may be less.
- Participants sign a consent form to acknowledge their rights, etc.
- **CONTROL GROUP ONLY:**

  Aside from the general orientation, the first hour or less will be focused on outlining expectations on how participants should document their comprehension gained from the completion of the tasks.

  The participants will also be briefed on what they or may not do (actions, tools used, questions asked) during the course of task completion. Specifically, participants will have full use of the computer and access to the internet, but their activities will be focused on comprehension using the Eclipse development environment and code browsing. Questions asked of the experimenter(s) will be restricted to those of a procedural nature, as opposed to those regarding content or help with completion of the tasks themselves.

  Participants will be encouraged to work closely and communicate with their partners on all aspects of the task completion in any way they see fit.

  It is anticipated that the second hour will be used for the completion of the given tasks, but participants may decide to or simply finish in a smaller amount of time.
- **EXPERIMENTAL GROUP ONLY:**

  Aside from the general orientation, the first hour or less will be focused on outlining expectations on how participants should document their comprehension gained from the completion of the tasks using *Pollinator*.

The participants will also be briefed on what they or may not do (actions, tools used, questions asked) during the course of task completion. Specifically, participants will have full use of the computer and access to the internet, but their activities will be focused on comprehension using *Pollinator* within the Eclipse development environment and code browsing. Questions asked of the experimenter(s) will be restricted to those of a procedural nature, as opposed to those regarding content or help with completion of the tasks themselves.

Participants will be encouraged to work closely and communicate with their partners on all aspects of the task completion, hopefully taking advantage of the features of *Pollinator* where they deem it advantageous.

It is anticipated that the second hour will be used for the completion of the given tasks, but participants may decide to or simply finish in a smaller amount of time.

Training

CONTROL GROUP:

1. Familiarize (if necessary) with Eclipse concepts such as navigation (opening files), code browsing, perspectives.
2. Show using sample system (mock Perl Interpreter) and two sample tasks.
3. Give a Brief overview of the subject e-voting system, its purpose and show some examples of its interface using Anjan's thesis outline. Do NOT show any examples of the code fragments responsible for designated functionality.
4. Ask some orientation questions, then explain a bit about the concepts if needed: ask if worked with JDBC or database access in java before, explain the JSP to Java translation if necessary.

EXPERIMENTAL GROUP:

1. Familiarize (if necessary) with Eclipse concepts such as navigation (opening files, seeing class methods, views), code browsing, perspectives.
2. Introduce the basic features of Pollinator, using the Pollinator tutorial document.
3. Show how to complete tasks with Pollinator given a sample system (mock Perl Interpreter) and two sample tasks.

4. Give a Brief overview of the subject e-voting system, its purpose and show some examples of its interface using Anjan's thesis outline. Do NOT show any examples of the code fragments responsible for designated functionality.

5. Ask some orientation questions, then explain a bit about the concepts if needed: ask if worked with JDBC or database access in java before, explain the JSP to Java translation if necessary.

Task Completion

During this portion of the experiment session, the pair under observation will be given tasks, one at a time, until the set of tasks has been completed or until an hour has elapsed.

Only one task is given as they are completed in order to reduce bias that may be introduced by the participants looking in advance at later tasks and incorporating this knowledge into their completion of the current tasks. This also ensures that the tasks are completed in a controlled order, which brings more consistency to the comparisons to be drawn between each study instance.

Post-Study Questionnaire

In this part of the experiment, the participants will be asked to complete a short post-study questionnaire regarding their experiences and opinions on using *Pollinator*.

Debriefing and Thanks

Here we reiterate the goals and objectives of the study and give participants the opportunity to have themselves excluded if they feel uncomfortable or for whatever reason decide to decline participation.

The particpants are thanked for the involvement and encouraged to contact the study author in the future with any questions or concerns.

A small token of appreciation (honorarium) is given to the participants along with a sheet they sign acknowledging receipt of the honorarium.

B.3.3 Restrictions

- No hints to be given to participants at any point as to any answers or solutions to the tasks given.

- Only questions about use of tools and the conduct of the study itself should be answered prior to the debriefing – nothing about how to complete or answer the questions in terms of content

B.3.4 History

- May 14, 2007 - Revision 0.1
- June 6, 2007 - Revision 0.2: *added questionnaire and honorarium information*

B.4 Pilot Tasks

Name 1: ______________________________Name 2: ______________________________

TASKS FOR PROGRAM COMPREHENSION STUDY

The software system you are studying today is an implementation of a web-based system for administering and conducting elections. Administrators may use the system to register users (voters), create elections and candidates, and perform other maintenance tasks. Voters may access the system to register their vote with an election. The administrator and voter accounts are based on a username-password mechanism.

The system is composed of several JSP and Java files. For the purposes of this study, the translated-to-Java JSP files should be looked at, as opposed to the HTML/JSP precursors.

Please answer, as best you can, the following questions regarding the online voting system. You may frame the structure of the Pollinator comprehension tree and questions however you see fit.

1. Using Pollinator, document those classes which represent the main business objects of the system.

Please use Pollinator to document your final answer to the question

2. Using Pollinator, document the sequence of method invocations in the registration of a new voter, starting from the administration page (add_voter_jsp.java).

Please use Pollinator to document your final answer to the question

3. Using Pollinator, document 'where' (i.e. which classes, methods, code locations, etc) the database access is implemented for the main business objects you documented earlier. Database "access" is defined as locations where data is directly read from or written to the database.

Please use Pollinator to document your final answer to the question

4. Where does authorization for administrator login occur? (i.e. which .jsp file / class and line(s) of code).

Please use Pollinator to document your final answer to the question

5. What does the function loadAllPosts() do? (post-conditions / effects of its execution.)

Please use Pollinator to document your final answer to the question

B.5 Control Tasks

Name 1: ______________________________ Name 2: ______________________________

TASKS FOR PROGRAM COMPREHENSION STUDY

The software system you are studying today is an implementation of a web-based system for administering and conducting elections. Administrators may use the system to register users (voters), create elections and candidates, and perform other maintenance tasks. Voters may access the system to register their vote with an election. The administrator and voter accounts are based on a username-password mechanism.

The system is composed of several JSP and Java files. For the purposes of this study, the translated-to-Java JSP files should be looked at, as opposed to the HTML/JSP precursors.

Please answer, as best you can, the following questions regarding the online voting system.

1. Document those classes which represent the main business or domain objects of the system. From Wikipedia: "Business objects are objects in an object-oriented computer program that abstract the entities in the domain that the program is written to represent."

Please use the space below to document your final answer to the question.

2. Document the sequence of method invocations in the registration of a new voter, at the level of the administration page (add_voter_jsp.java).

Please use the space below to document your final answer to the question.

3. Document 'where' (i.e. which classes, methods, code locations, etc) the database access is implemented for three (3) of the main business objects you documented earlier. Database "access" is defined as locations where data is directly read from or written to the database.

Please use the space below to document your final answer to the question.

4. Where does authorization for administrator login occur? (i.e. which .jsp file / class and line(s) of code).

Please use the space below to document your final answer to the question.

5. What does the function loadAllPosts() do? (post-conditions / effects of its execution.) Please also examine the effects of the findPost() method that is called.

Please use the space below to document your final answer to the question.

END OF TASKS

B.6 Experimental Tasks

Name 1: ______________________________ Name 2: ______________________________

TASKS FOR PROGRAM COMPREHENSION STUDY

The software system you are studying today is an implementation of a web-based system for administering and conducting elections. Administrators may use the system to register users (voters), create elections and candidates, and perform other maintenance tasks. Voters may access the system to register their vote with an election. The administrator and voter accounts are based on a username-password mechanism.

The system is composed of several JSP and Java files. For the purposes of this study, the translated-to-Java JSP files should be looked at, as opposed to the HTML/JSP precursors.

Please answer, as best you can, the following questions regarding the online voting system. You may frame the structure of the Pollinator comprehension tree and questions however you see fit.

1. Using Pollinator, document those classes which represent the main business or domain objects of the system. From Wikipedia: "Business objects are objects in an object-oriented computer program that abstract the entities in the domain that the program is written to represent."

Please use Pollinator to document your final answer to the question

2. Using Pollinator, document the sequence of method invocations in the registration of a new voter, at the level of the administration page (add_voter_jsp.java).

Please use Pollinator to document your final answer to the question

3. Using Pollinator, document 'where' (i.e. which classes, methods, code locations, etc) the database access is implemented for three (3) of the main business objects you documented earlier. Database "access" is defined as locations where data is directly read from or written to the database.

Please use Pollinator to document your final answer to the question

4. Where does authorization for administrator login occur? (i.e. which .jsp file / class and line(s) of code).

Please use Pollinator to document your final answer to the question

5. What does the function loadAllPosts() do? (post-conditions / effects of its execution.)
   Please also examine the effects of the findPost() method that is called.

Please use Pollinator to document your final answer to the question

END OF TASKS

B.7 Pre-Study Questionnaire

Pre-Study Questionnaire for Pollinator User Study

For each multiple choice question, please choose ONE answer.

Name: ______________________________ E-mail: ______________________________

1. How much experience do you have working with the Java programming language?

- None
- 1-2 years
- 3-4 years
- 5+ years

2. How would you rate your skill level in the Java programming language?

- Poor
- Novice
- Average
- Expert

3. How would you rate your familiarity with the usage of the Eclipse development environment?

- I use it frequently or almost every day.
- I use it occasionally and am familiar with basic usage.
- I have only used Eclipse once or twice.
- I have never used Eclipse.

B.8 Post-Study Questionnaire (EXPERIMENTAL)

For each multiple choice question, please choose ONE answer.

If extra space is needed for the long answer questions, please write on the back of the paper.

1. How difficult did you find the program comprehension task(s)?

   a) very easy
   b) easy
   c) average
   d) difficult
   e) very difficult

2. Did you find that the Pollinator tool helped in the completion of the programming task(s)? Why or why not?

3. Did Pollinator help coordinate communication between you and your partner? Why or why not?

4. How usable (ease of use, clarity of action/consequence pairing, UI) is Pollinator?

   a) very bad
   b) bad
   c) average
   d) good
   e) very good

5. What did you like most about Pollinator? Why?

6. What did you least like about Pollinator? Why?

7. Would you use Pollinator for your own work and/or research?

8. Do you have any other feedback or suggestions?

B.9 Post-Study Questionnaire (CONTROL)

Name: ____

For each multiple choice question, please choose ONE answer.

If extra space is needed for the long answer questions, please write on the back of the paper.

1. How difficult did you find the program comprehension task(s)?

   a) very easy
   b) easy
   c) average
   d) difficult
   e) very difficult

2. Do you think the method of documenting program comprehension you used in this study is reliable and useful? Why or why not?

3. How did this method of documenting program comprehension affect the coordination of communication between you and your partner, if at all?

4. Do you think a specialized tool could help in the completion of the program comprehension task(s)? Why or why not?

5. Could a tool help coordinate communication between you and your partner? If so, how?

6. Do you have any additional feedback or general comments?

B.10 Honorarium Acknowledgement

Acknowledgement of receipt of honorarium

Re: Pollinator User Study

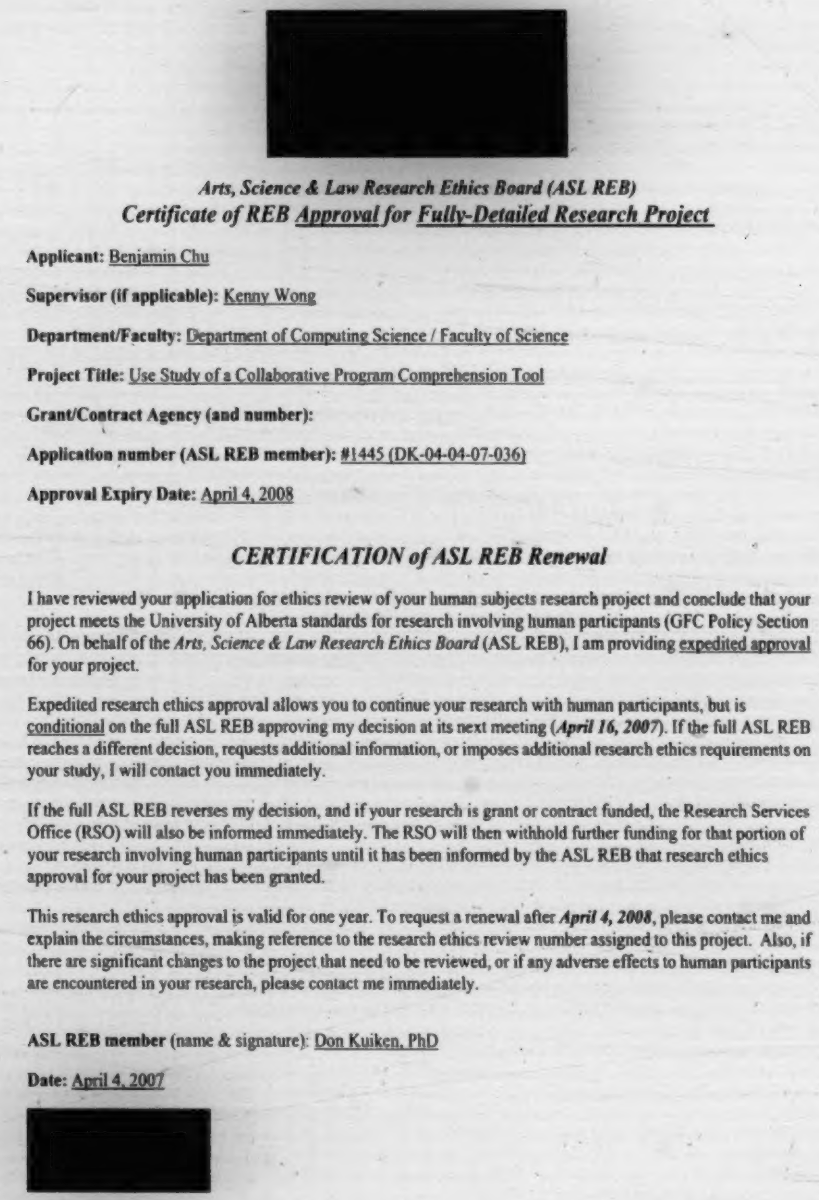
I, ______________________, acknowledge that I have received an honorarium of a Future Shop gift card (value of $20) as a token of appreciation for participation in the Pollinator user study conducted by Benjamin Chu as part of his Master of Science research under the supervision of Dr. Kenny Wong.

Signed,

Signature: ______________________

Date: ______________________

B.11 Ethics Approval Certificate



[redacted]

Arts, Science & Law Research Ethics Board (ASL REB)

Certificate of REB Approval for Fully-Detailed Research Project

Applicant: Benjamin Chu

Supervisor (if applicable): Kenny Wong

Department/Faculty: Department of Computing Science / Faculty of Science

Project Title: Use Study of a Collaborative Program Comprehension Tool

Grant/Contract Agency (and number):

Application number (ASL REB member): #1445 (DK-04-04-07-036)

Approval Expiry Date: April 4, 2008

CERTIFICATION of ASL REB Renewal

I have reviewed your application for ethics review of your human subjects research project and conclude that your project meets the University of Alberta standards for research involving human participants (GFC Policy Section 66). On behalf of the *Arts, Science & Law Research Ethics Board* (ASL REB), I am providing expedited approval for your project.

Expedited research ethics approval allows you to continue your research with human participants, but is conditional on the full ASL REB approving my decision at its next meeting (***April 16, 2007***). If the full ASL REB reaches a different decision, requests additional information, or imposes additional research ethics requirements on your study, I will contact you immediately.

If the full ASL REB reverses my decision, and if your research is grant or contract funded, the Research Services Office (RSO) will also be informed immediately. The RSO will then withhold further funding for that portion of your research involving human participants until it has been informed by the ASL REB that research ethics approval for your project has been granted.

This research ethics approval is valid for one year. To request a renewal after ***April 4, 2008***, please contact me and explain the circumstances, making reference to the research ethics review number assigned to this project. Also, if there are significant changes to the project that need to be reviewed, or if any adverse effects to human participants are encountered in your research, please contact me immediately.

ASL REB member (name & signature): Don Kuiken, PhD

Date: April 4, 2007

[redacted]

Appendix C

User Study Task Analysis

This appendix contains a detailed transcript of the actions of the participants of the *Pollinator* user study as they completed the program comprehension tasks given to them. The documented answers given by the participants is also included alongside the reference answers.

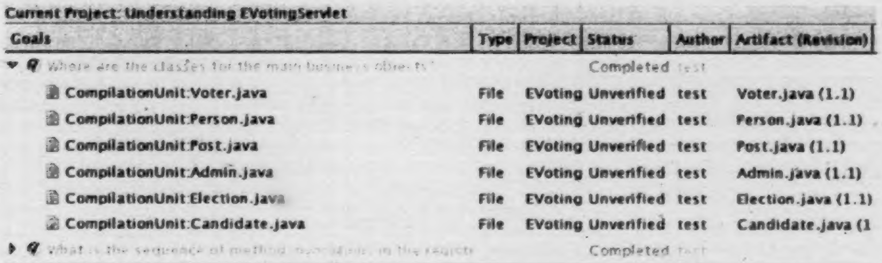
The text of the tasks in this section is as they were presented to the participants in the control and experimental studies. Any changes from the pilot study phrasing are detailed in footnotes to the task descriptions.

Task 1

> *Document those classes which represent the main business or domain objects of the system. From Wikipedia: "Business objects are objects in an object-oriented computer program that abstract the entities in the domain that the program is written to represent."* [1]

Pilot Study

1. The participants create a goal "What are the main business object classes"?
2. They look through the `.java` files of the project to get an impression of what methods are in the various classes.
3. Now they take a closer look at file contents to judge their importance.
4. Files are dragged and dropped to the single main goal (that represent business objects). The participants also wanted to add the `com.polling.admin package`, but *Pollinator* does not allow this.
5. The goal is marked as completed.



Current Project: Understanding EVotingServlet

Goals	Type	Project	Status	Author	Artifact (Revision)
Where are the classes for the main business objects?			Completed	test	
CompilationUnit:Voter.java	File	EVoting	Unverified	test	Voter.java (1.1)
CompilationUnit:Person.java	File	EVoting	Unverified	test	Person.java (1.1)
CompilationUnit:Post.java	File	EVoting	Unverified	test	Post.java (1.1)
CompilationUnit:Admin.java	File	EVoting	Unverified	test	Admin.java (1.1)
CompilationUnit:Election.java	File	EVoting	Unverified	test	Election.java (1.1)
CompilationUnit:Candidate.java	File	EVoting	Unverified	test	Candidate.java (1
What is the sequence of method invocations in the regist			Completed	test	

Figure C.1: Pilot study task 1 result

Control Study

---

[1] The Wikipedia definition was not present in the pilot study.

1. The participants read the task and go to Eclipse to look at the Java classes. They also discuss what a business object is.

2. They look at the `Candidate` methods, C2 discusses his usual methodology for understanding code and explains some Eclipse conventions to his partner, such as the icons used for denoting method visibility and type.

3. C1 writes some notes down on what they are looking at, they decide that `Candidate` should be included as a business object and C2 waits for C1 to finish writing.

4. The second object written down is `Election`; they look through the code and see some database access code.

5. Now, looking at `Post`, it is written down. C2 asks about the role of `Post` in the system. At this point, it appears that the two teammates may not be in sync with each other's mental context.

6. Now `Voter` is added.

7. (Note: C2 controls the computer, both mouse and keyboard).

8. They go back to `Post` and do a file search for "candidate", hoping to find some relationship to further their understanding.

9. (Note: C2, Eclipse expert, knows some tricks, such as ctrl-clicking on variable name to go to its definition or declaration).

10. They discuss that probably they don't need to go into the amount of detail they are looking at for this particular task and then ask a question about what the word "Document" means in the task wording. Experimenter replies it is related to making note of something, essentially writing down what is asked of in the task.

11. When writing, C2 suggests to C1 that the business objects be grouped into the packages in which they are found.

12. They now go to look at `Person` and decide not to include it based on C2's opinion that a business object is something that is "domain-specific", something that he feels `Person` is not one of.

13. They check and decide that other classes in `com` package are irrelevant.

14. Now some time is spent looking through `Admin`, and they decide it is also a business object because it has election-related code inside.

15. The `.jsp` files are quickly scanned (the names) and skipped as potential business object candidates.

`polling`
1) Candidate.java
2) Election.java
3) Post.java
4) Voter.java

`com`
1) Admin.java

Figure C.2: Control study answer to Task 1

Experimental Study

1. There is discussion first on where to begin, so the participants are not even looking at the computer. The discussion is at the domain level of elections and its entities.

2. They then decide that further debate / discussion on this topic might be fruitless, so they go to the computer and create a new goal. However, they have trouble finding how to create a goal since a project is not opened, and need prompting from the experimenter to help in this technical problem.

3. They now search through goal templates but find nothing applicable and just create the goal as "What are the main business objects?"

4. They look through `.java` files in the `com` package, to see if there's anything relevant. In particular, they open up `Admin` for inspection.

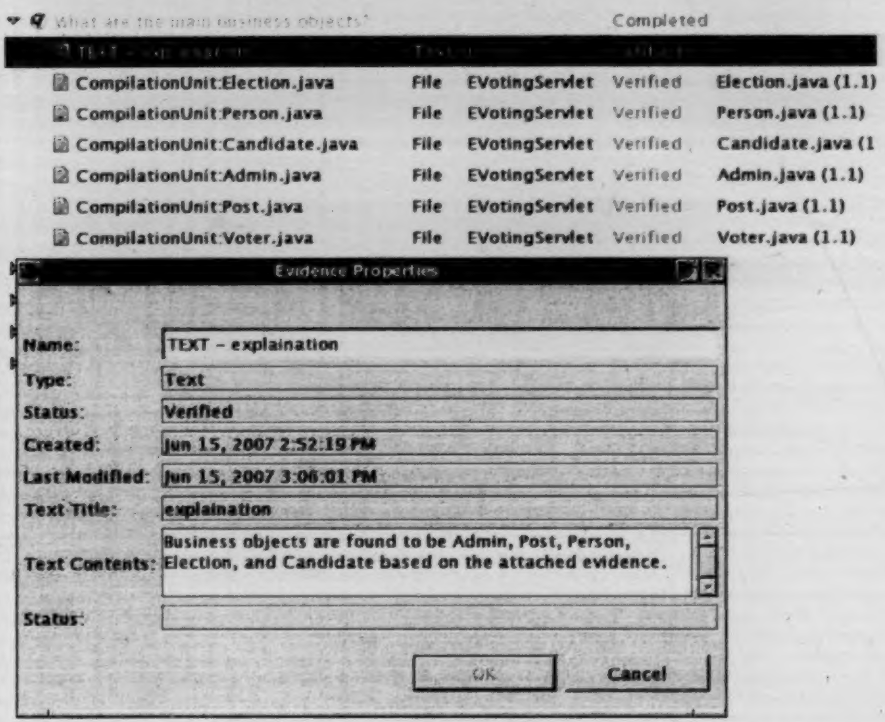
5. E1 speculates that if entity beans are used, they are usually the business objects.

6. `Admin.java` is added to the main goal.

7. They now go through the `.java` classes adding those they suspect are business objects based on the domain and a brief peek at the file contents and exclude those that are action/utility/exception classes (so they add `Candidate`, `Post`, `Voter`, `Election`) and skip the `.jsp` files. E2 writes these down as they go along. E2

also has an interest in knowing the relationship between classes for a better basis of understanding.

8. Now, after deliberation, they decide to add `Person`, and expect some sort of object hierarchy.
9. The goal is marked completed, and all evidence marked as verified.
10. As an afterthought, an explanation is added to clearly and explicitly answer the question through the use of attaching text evidence.



What are the main business objects? Completed

CompilationUnit:Election.java	File	EVotingServlet	Verified	Election.java (1.1)
CompilationUnit:Person.java	File	EVotingServlet	Verified	Person.java (1.1)
CompilationUnit:Candidate.java	File	EVotingServlet	Verified	Candidate.java (1
CompilationUnit:Admin.java	File	EVotingServlet	Verified	Admin.java (1.1)
CompilationUnit:Post.java	File	EVotingServlet	Verified	Post.java (1.1)
CompilationUnit:Voter.java	File	EVotingServlet	Verified	Voter.java (1.1)

Evidence Properties

Name: TEXT – explaination
Type: Text
Status: Verified
Created: Jun 15, 2007 2:52:19 PM
Last Modified: Jun 15, 2007 3:06:01 PM
Text Title: explaination
Text Contents: Business objects are found to be Admin, Post, Person, Election, and Candidate based on the attached evidence.
Status:

Figure C.3: Experimental study task 1 result

Task 2

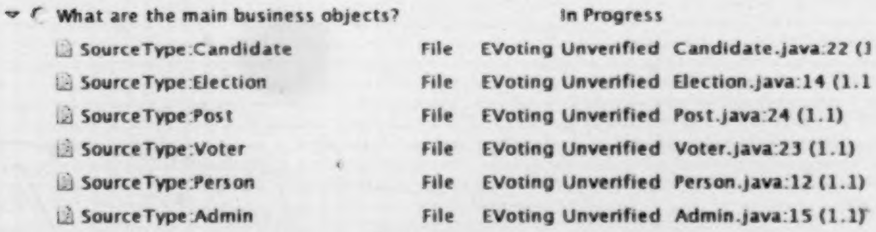
Document the sequence of method invocations in the registration of a new voter, at the level of the administration page (`add_voter_jsp.java`*).* [2]

Pilot Study

1. A new goal is created as "What is the sequence of method invocations?"

---

[2] The word "starting" was used instead of "level" in the pilot.



What are the main business objects?		In Progress	
SourceType:Candidate	File	EVoting Unverified	Candidate.java:22 (1
SourceType:Election	File	EVoting Unverified	Election.java:14 (1.1
SourceType:Post	File	EVoting Unverified	Post.java:24 (1.1)
SourceType:Voter	File	EVoting Unverified	Voter.java:23 (1.1)
SourceType:Person	File	EVoting Unverified	Person.java:12 (1.1)
SourceType:Admin	File	EVoting Unverified	Admin.java:15 (1.1)

Figure C.4: Reference answer for task 1

2. The participants open the `add_voter_jsp.java` file and maximize the editor view.

3. They scroll through the file, looking for the relevant methods.

4. They find the `newvoter` variable and do a search in the file for previous instances, to see where it is declared and where it is used.

5. Next, they discuss if instantiation and/or use of beans methods should be included.

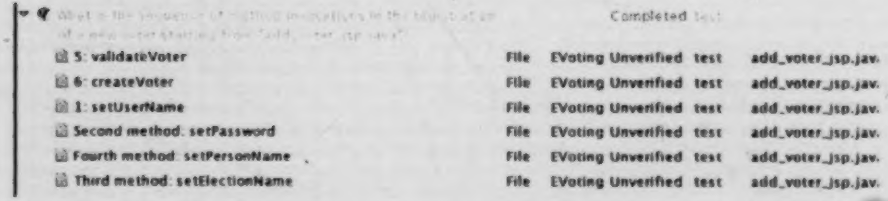
6. They check the rest of the file, after the `newPerson.setX()` method calls.

7. The editor view is restored, and they scan the file again searching for `newvoter`.

8. The participants try to drag the selected method code, but this of course is unsuccessful.

9. The "Add as evidence" context menu item is used instead.

10. The method evidence is labeled as "first", "second", "third", etc. with the method name included (they then rename some of them to numbers in trying to get them sorted properly, but this does not work).



		Completed		
5: validateVoter	File	EVoting Unverified	test	add_voter_jsp.jav.
6: createVoter	File	EVoting Unverified	test	add_voter_jsp.jav.
1: setUserName	File	EVoting Unverified	test	add_voter_jsp.jav.
Second method: setPassword	File	EVoting Unverified	test	add_voter_jsp.jav.
Fourth method: setPersonName	File	EVoting Unverified	test	add_voter_jsp.jav.
Third method: setElectionName	File	EVoting Unverified	test	add_voter_jsp.jav.

Figure C.5: Pilot study task 2 result

Control Study

1. The participants navigate to the `.jsp` file, `add_voter_jsp.java` and take a look through it.

2. They reason quickly that there is code in the file for initializing a new `Voter`.

3. C2 asks for a clarification on the meaning of the phrase in the task "at the level of".

4. Further code browsing is done, and the participants are fairly silent.

5. They then find the section of code for registration on the screen and scroll a bit past it to the page rendering code. They then scroll back up to the top.

6. They spend some time rereading the code for initializing `newvoter` and notice that a new `Person` is also created in the process.

7. (Note: a lot of context seems to be maintained in the minds of the individual participants, which apparently necessitates reorientation and reinforcement of previous understanding by going over previously seen code more than once).

8. Now, C1 starts to write down the method invocations, starting with `pageContext.getAttributes()`.

9. C2 now asks if there is a business rule for registering a new voter, and the experimenter shows the screenshot that was used in the training phase of the experiment depicting the screen for new voter registration by an administrator.

10. C1 then writes down the rest of the method invocations as C2 scrolls and reads through them on-screen. This ends with `Voter.addFieldError()` in the `catch` clause.

11. They perform a quick scan of the rest of the `.jsp` file and decide they do not need to bother with it.

Experimental Study

1. The task is read and templates looked at but no suitable template found, so they just create a goal as "What is the method invocation sequence in the registration of a new voter?".

2. The participants add `add_voter_jsp.java` right away to the goal.

New Voter

- `getAttribute`
- `instantiate`
- `setAttribute`

Election

- `getAttribute`
- `instantiate`
- `setAttribute`

new Person

- `getAttribute`
- `instantiate`
- `setAttribute`

```
setUserName()
setPassword()
setElectionName()
setPersonName()
validateVoter()
createVoter()
addFieldError()
```

Figure C.6: Control study answer to Task 2

3. Now they perform a file search in `add_voter_jsp.java` for "register" and find nothing.
4. They scan through the file by eyeball quickly to see if there's anything noticeable.
5. Then they go through again more slowly, to get a general idea of the code structure.
6. They find the `newvoter` variable initialization and discuss this in detail.
7. E2 writes down a note, his idea of the first step in the registration but this behavior is later abandoned in favor of discussion and use of *Pollinator*.
8. They go back to the `Voter` class and see which fields need to be initialized when registering a voter, and E2 writes these down.
9. The `add_voter_jsp.java` is searched to see if any `Voter` methods are called from there and find one to `Voter.createVoter()` and decide this is the place.
10. Now they decide what to attach, which is just the specific range of code.
11. The `createVoter()` method itself is attached as evidence, but E2 doubts that this is necessary.
12. A text explanation is attached to explicitly spell out the method sequence.
13. Next, the goal and evidence are marked as completed / verified.

Task 3

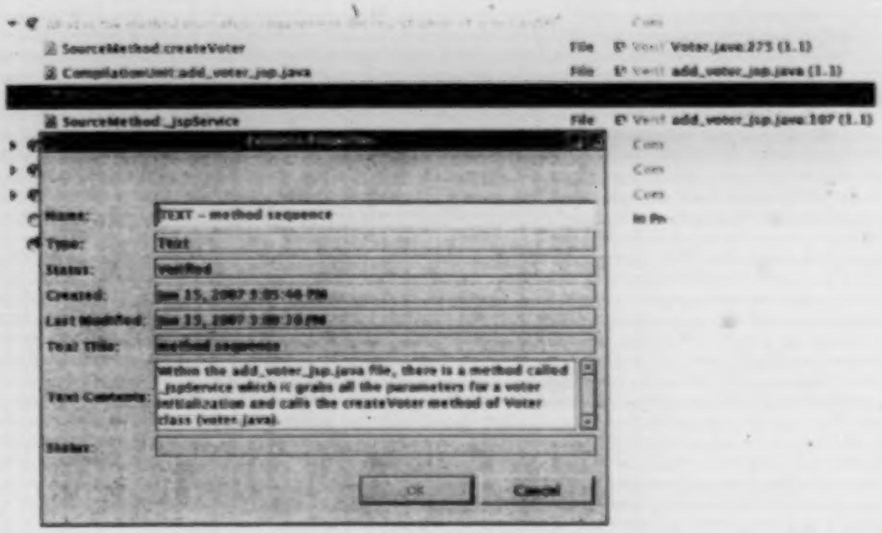
> *Document 'where' (i.e. which classes, methods, code locations, etc) the database access is implemented for three (3) of the main business objects you documented earlier. Database "access" is defined as locations where data is directly read from or written to the database.* [3]

Pilot Study

1. The participants create a new goal right away, an action that appears to be second nature at this point.
2. They spend a bit of time discussing the task before filling out the goal dialog.
3. Now, they look at the business objects from the first task

---

[3] The number of business objects was not limited in the pilot.



SourceMethod:createVoter File Verif Voter.java:275 (1.1)

CompilationUnit:add_voter_jsp.java File Verif add_voter_jsp.java (1.1)

SourceMethod:_jspService File Verif add_voter_jsp.java:107 (1.1)

Name: TEXT – method sequence

Type: Text

Status: verified

Created: Jun 15, 2007 3:05:46 PM

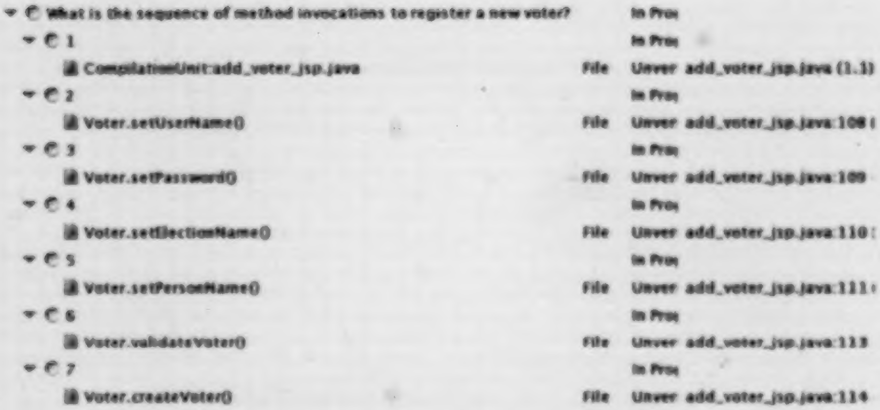
Last Modified: Jun 15, 2007 3:09:10 PM

Text Title: method sequence

Text Contents: Within the add_voter_jsp.java file, there is a method called _jspService which it grabs all the parameters for a voter initialization and calls the createVoter method of Voter class (voter.java).

Status:

Figure C.7: Experimental study task 2 result



What is the sequence of methods invocations to register a new voter? In Proj

1 In Proj

CompilationUnit:add_voter_jsp.java File Unver add_voter_jsp.java (1.1)

2 In Proj

Voter.setUserName() File Unver add_voter_jsp.java:108

3 In Proj

Voter.setPassword() File Unver add_voter_jsp.java:109

4 In Proj

Voter.setElectionName() File Unver add_voter_jsp.java:110

5 In Proj

Voter.setPersonName() File Unver add_voter_jsp.java:111

6 In Proj

Voter.validateVoter() File Unver add_voter_jsp.java:113

7 In Proj

Voter.createVoter() File Unver add_voter_jsp.java:114

Figure C.8: Reference answer for task 2

4. They look through the `.java` files for `Admin`, to *find that business object's database* access code.

5. Next, they highlight and attach the discovered `Admin` code to a newly created subgoal of main goal (there is a bit of trouble figuring out how to do this based on the goal selection dialog).

6. This is repeat (except for the creation of a new goal) for the rest of the `Admin` database code.

7. They create subgoal explicitly for the next business object, then repeat as in step 4.

8. Steps 5 to 6 are repeated for the rest of the business objects.

9. The goals are marked as completed.

Control Study

1. C2 directs the group to go to the `SQLQueries.properties` file to check out the types of queries inside that would be associated with business object database access. The first object searched for is `Candidate`-related queries.

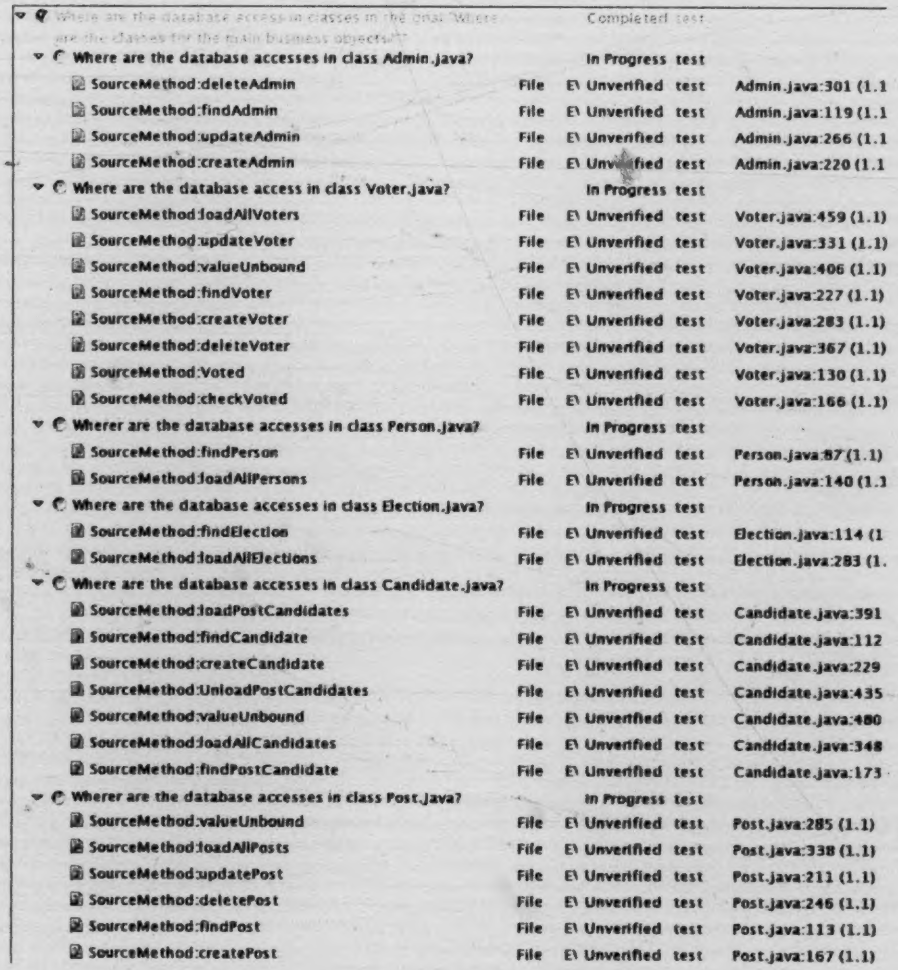
2. They decide to do a reverse lookup of the code by searching for references to the query strings stored in the `SQLQueries.properties` bundle that are used in the code.

3. C1 is acting as an extra set of eyes, pointing out relevant things for both partners to notice.

4. They decide on answering the task for the following three business objects: `Candidate`, `Election` and `Voter`.

5. First, C1 writes down "Candidate" and their plan is to proceed to look for occurrences of `Candidate`-related database query occurrences in `SQLQueries.properties`. The file itself is modified so that visually, they can group together those queries that are associated with a particular business object.

6. C2 uses the Eclipse global file search to look for query occurrences in the code by searching for query string identifiers from the `SQLQueries.properties` file.



Where are the database access in classes in the goal "Where are the classes for the main business objects?"		Completed	test	
Where are the database accesses in class Admin.java?		In Progress	test	
SourceMethod:deleteAdmin	File	E\ Unverified	test	Admin.java:301 (1.1
SourceMethod:findAdmin	File	E\ Unverified	test	Admin.java:119 (1.1
SourceMethod:updateAdmin	File	E\ Unverified	test	Admin.java:266 (1.1
SourceMethod:createAdmin	File	E\ Unverified	test	Admin.java:220 (1.1
Where are the database access in class Voter.java?		In Progress	test	
SourceMethod:loadAllVoters	File	E\ Unverified	test	Voter.java:459 (1.1)
SourceMethod:updateVoter	File	E\ Unverified	test	Voter.java:331 (1.1)
SourceMethod:valueUnbound	File	E\ Unverified	test	Voter.java:406 (1.1)
SourceMethod:findVoter	File	E\ Unverified	test	Voter.java:227 (1.1)
SourceMethod:createVoter	File	E\ Unverified	test	Voter.java:283 (1.1)
SourceMethod:deleteVoter	File	E\ Unverified	test	Voter.java:367 (1.1)
SourceMethod:Voted	File	E\ Unverified	test	Voter.java:130 (1.1)
SourceMethod:checkVoted	File	E\ Unverified	test	Voter.java:166 (1.1)
Wherer are the database accesses in class Person.java?		In Progress	test	
SourceMethod:findPerson	File	E\ Unverified	test	Person.java:87 (1.1)
SourceMethod:loadAllPersons	File	E\ Unverified	test	Person.java:140 (1.1
Where are the database accesses in class Election.java?		In Progress	test	
SourceMethod:findElection	File	E\ Unverified	test	Election.java:114 (1
SourceMethod:loadAllElections	File	E\ Unverified	test	Election.java:283 (1.
Where are the database accesses in class Candidate.java?		In Progress	test	
SourceMethod:loadPostCandidates	File	E\ Unverified	test	Candidate.java:391
SourceMethod:findCandidate	File	E\ Unverified	test	Candidate.java:112
SourceMethod:createCandidate	File	E\ Unverified	test	Candidate.java:229
SourceMethod:UnloadPostCandidates	File	E\ Unverified	test	Candidate.java:435
SourceMethod:valueUnbound	File	E\ Unverified	test	Candidate.java:480
SourceMethod:loadAllCandidates	File	E\ Unverified	test	Candidate.java:348
SourceMethod:findPostCandidate	File	E\ Unverified	test	Candidate.java:173
Wherer are the database accesses in class Post.java?		In Progress	test	
SourceMethod:valueUnbound	File	E\ Unverified	test	Post.java:285 (1.1)
SourceMethod:loadAllPosts	File	E\ Unverified	test	Post.java:338 (1.1)
SourceMethod:updatePost	File	E\ Unverified	test	Post.java:211 (1.1)
SourceMethod:deletePost	File	E\ Unverified	test	Post.java:246 (1.1)
SourceMethod:findPost	File	E\ Unverified	test	Post.java:113 (1.1)
SourceMethod:createPost	File	E\ Unverified	test	Post.java:167 (1.1)

Figure C.9: Pilot study task 3 result

7. C1 then writes down the class, method, and code location (line range) for each database access implementation. This is repeated for all the database access associated with a business object (i.e. first search for the query, then find occurrences in the code, C2 recites the name and location information, and C1 writes it down). When the next business object is up, the `SQLQueries.properties` is modified to group potential candidates for query strings.

CANDIDATE
1) Class - Candidate
Method - findCandidate
Code location - line 104 → line 156

2) Method - findPostCandidate
code loc → line 165 → line 219

3) createCandidate()
line 222 → line 268

4) loadAllCandidates()
line 341 → line 380

5) loadPostCandidates()
line 384 → line 425

Election
Class - Election

1) Method - findElection()
line 105 → line 159

2) loadAllElections()
line 276 → line 316

(a)

Voter
Class - Voter

1) findVoter()
line 219 → line 274

2) checkVoted()
line 160 → 216

3) createVoter()
line 276 → line 326
4) updateVoter()
line 328 → line 362

5) Voted()
line 127 → 158

6) deleteVoter()
line 364 → line 395

7) loadAllVoters()
line 452 → line 491

(b)

Figure C.10: Control study answer to Task 3

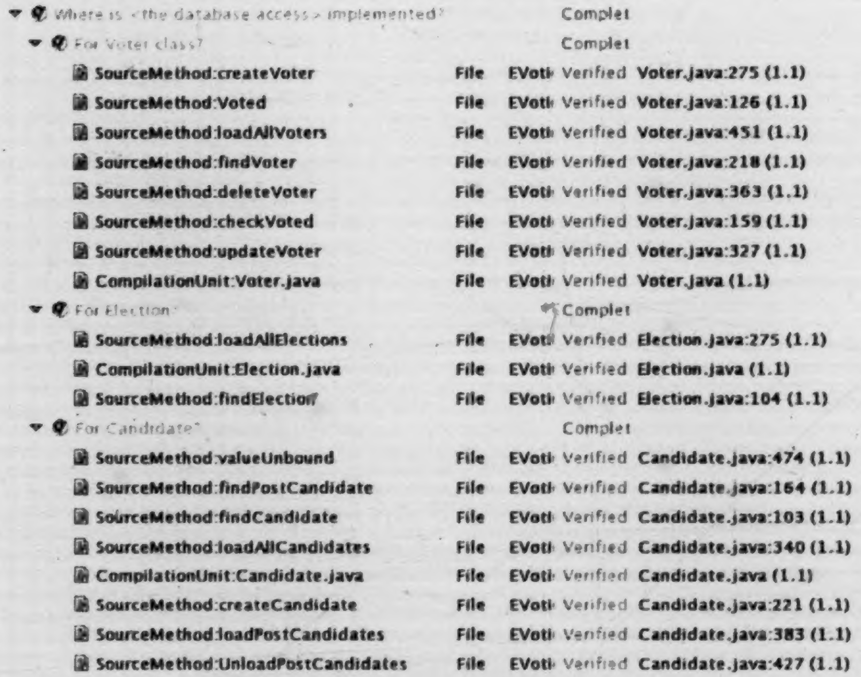
Experimental Study

1. Participants can use a template now to create the goal. E2 seems to stop note-taking at this point.

2. An interface issue arises, since the pair did not notice the "root goal" radio button to switch from their accidentally created subgoal, instead they started again and created

a goal as “Where is the database access implemented?”.

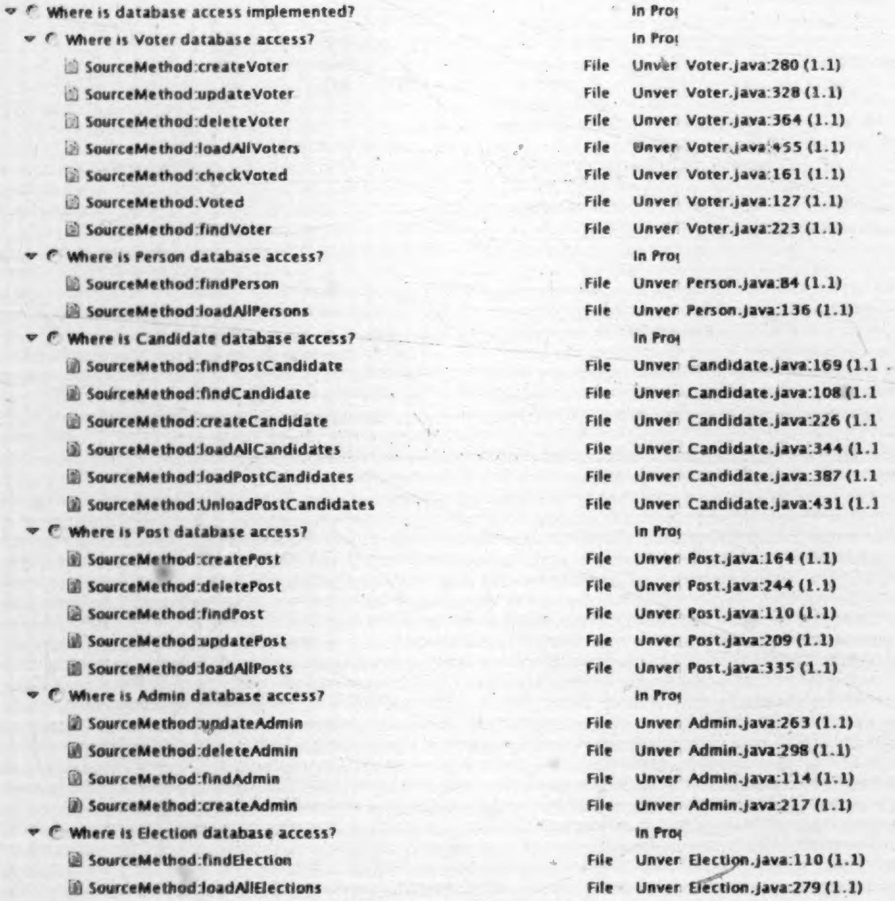
3. They decide to look at `Voter` first and its method signatures and discuss what needs to be looked at.
4. First, `Voter.java` is added as evidence to main goal.
5. Now they search through the methods of `Voter` looking for database access, adding the methods to the goal as found by dragging from the “Members” view.
6. When done for `Voter`, they decide to create a subgoal for each business object and then attach evidence to the specific subgoals. For example the subgoal for `Voter` is “For voter class?” and this is repeated for the remaining two classes (`Election` and `Candidate`).
7. As they are going through the code, they notice that quite a bit of it could probably be refactored as there is a lot of shared or common code.
8. Now, goals are marked completed, and evidence marked verified.



▼ Where is <the database access> implemented?			Complet	
▼ For Voter class?			Complet	
SourceMethod:createVoter	File	EVoti	Verified	Voter.java:275 (1.1)
SourceMethod:Voted	File	EVoti	Verified	Voter.java:126 (1.1)
SourceMethod:loadAllVoters	File	EVoti	Verified	Voter.java:451 (1.1)
SourceMethod:findVoter	File	EVoti	Verified	Voter.java:218 (1.1)
SourceMethod:deleteVoter	File	EVoti	Verified	Voter.java:363 (1.1)
SourceMethod:checkVoted	File	EVoti	Verified	Voter.java:159 (1.1)
SourceMethod:updateVoter	File	EVoti	Verified	Voter.java:327 (1.1)
CompilationUnit:Voter.java	File	EVoti	Verified	Voter.java (1.1)
▼ For Election?			Complet	
SourceMethod:loadAllElections	File	EVoti	Verified	Election.java:275 (1.1)
CompilationUnit:Election.java	File	EVoti	Verified	Election.java (1.1)
SourceMethod:findElection	File	EVoti	Verified	Election.java:104 (1.1)
▼ For Candidate?			Complet	
SourceMethod:valueUnbound	File	EVoti	Verified	Candidate.java:474 (1.1)
SourceMethod:findPostCandidate	File	EVoti	Verified	Candidate.java:164 (1.1)
SourceMethod:findCandidate	File	EVoti	Verified	Candidate.java:103 (1.1)
SourceMethod:loadAllCandidates	File	EVoti	Verified	Candidate.java:340 (1.1)
CompilationUnit:Candidate.java	File	EVoti	Verified	Candidate.java (1.1)
SourceMethod:createCandidate	File	EVoti	Verified	Candidate.java:221 (1.1)
SourceMethod:loadPostCandidates	File	EVoti	Verified	Candidate.java:383 (1.1)
SourceMethod:UnloadPostCandidates	File	EVoti	Verified	Candidate.java:427 (1.1)

Figure C.11: Experimental study task 3 result

Task 4



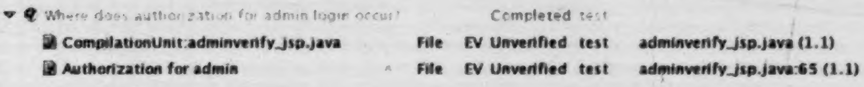
Where is database access implemented?		In Pro	
Where is Voter database access?		In Pro	
SourceMethod:createVoter	File	Unver	Voter.java:280 (1.1)
SourceMethod:updateVoter	File	Unver	Voter.java:328 (1.1)
SourceMethod:deleteVoter	File	Unver	Voter.java:364 (1.1)
SourceMethod:loadAllVoters	File	Unver	Voter.java:455 (1.1)
SourceMethod:checkVoted	File	Unver	Voter.java:161 (1.1)
SourceMethod:Voted	File	Unver	Voter.java:127 (1.1)
SourceMethod:findVoter	File	Unver	Voter.java:223 (1.1)
Where is Person database access?		In Pro	
SourceMethod:findPerson	File	Unver	Person.java:84 (1.1)
SourceMethod:loadAllPersons	File	Unver	Person.java:136 (1.1)
Where is Candidate database access?		In Pro	
SourceMethod:findPostCandidate	File	Unver	Candidate.java:169 (1.1
SourceMethod:findCandidate	File	Unver	Candidate.java:108 (1.1
SourceMethod:createCandidate	File	Unver	Candidate.java:226 (1.1
SourceMethod:loadAllCandidates	File	Unver	Candidate.java:344 (1.1
SourceMethod:loadPostCandidates	File	Unver	Candidate.java:387 (1.1
SourceMethod:UnloadPostCandidates	File	Unver	Candidate.java:431 (1.1
Where is Post database access?		In Pro	
SourceMethod:createPost	File	Unver	Post.java:164 (1.1)
SourceMethod:deletePost	File	Unver	Post.java:244 (1.1)
SourceMethod:findPost	File	Unver	Post.java:110 (1.1)
SourceMethod:updatePost	File	Unver	Post.java:209 (1.1)
SourceMethod:loadAllPosts	File	Unver	Post.java:335 (1.1)
Where is Admin database access?		In Pro	
SourceMethod:updateAdmin	File	Unver	Admin.java:263 (1.1)
SourceMethod:deleteAdmin	File	Unver	Admin.java:298 (1.1)
SourceMethod:findAdmin	File	Unver	Admin.java:114 (1.1)
SourceMethod:createAdmin	File	Unver	Admin.java:217 (1.1)
Where is Election database access?		In Pro	
SourceMethod:findElection	File	Unver	Election.java:110 (1.1)
SourceMethod:loadAllElections	File	Unver	Election.java:279 (1.1)

Figure C.12: Reference answer for task 3

Where does authorization for administrator login occur? (i.e. which .jsp file / class and line(s) of code).

Pilot Study

1. Participants open the `adminlogin_jsp.java` file.
2. Then the pair realize they should create a goal first before proceeding.
3. Now, they scroll through the file and find that it's not what they're expecting as it merely serves as a page for rendering the HTML form, but does no processing.
4. So now, attention is shifted to `adminverify_jsp.java` instead, and they browse through this file, seeing a declaration for an `Admin` object where the page form processing takes place after a submit is performed.
5. They drag `adminverify_jsp.java` to the goal.
6. Then they highlight the code where the authorization takes place and attach it to the goal as "Authorization for Admin".
7. They come back after finishing task five to mark this goal as completed.



Where does authorization for admin login occur?			Completed	test	
CompilationUnit:adminverify_jsp.java	File	EV	Unverified	test	adminverify_jsp.java (1.1)
Authorization for admin	File	EV	Unverified	test	adminverify_jsp.java:65 (1.1)

Figure C.13: Pilot study task 4 result

Control Study

1. They open `admin_login_jsp.java`.
2. C1 writes down this filename immediately but C2 says this is not where the actual authentication takes place, which is what the task is asking for.
3. So now they open up `adminverify_jsp.java` instead and look for where the database is checked for authorization of the administrator by looking through `SQLQueries.properties` again, finding the `findQuery` query string which seems to retrieve the stored password from the database.
4. It is seen that `findQuery` is referenced from `Admin.java` in the `findAdmin()` method which is in turn called from `adminverify_jsp.java`.

5. Finding the proper code location, they write down the information (`.jsp` file, class and line range of code).

JSP File → adminverify_jsp.java

Class → Admin.java

line 66 → line 69

Figure C.14: Control study answer to Task 4

Experimental Study

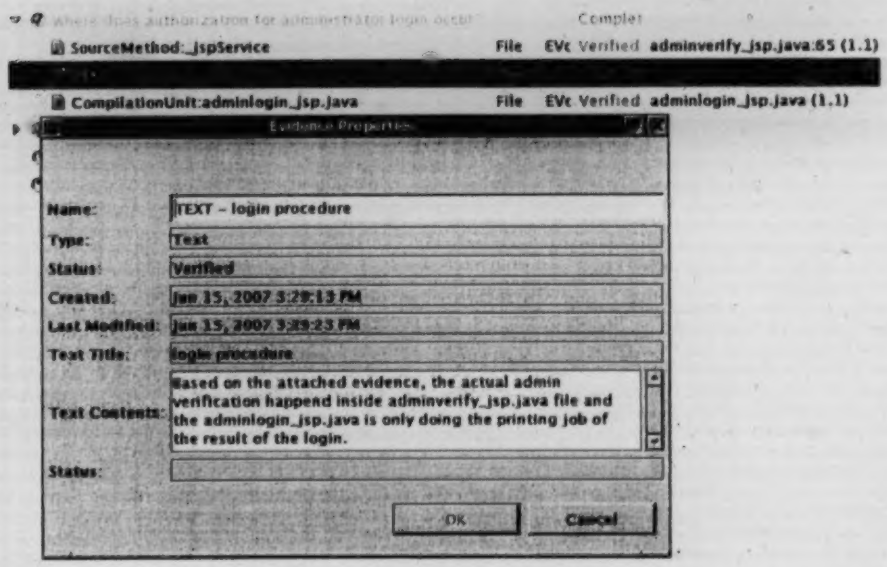
1. A new goal is created first as "Where does authorization for administrator login occur?" since no suitable template is found.
2. They jump into the `.jsp` files and look first at `adminlogin_jsp.java`, immediately attaching it as evidence to the goal.
3. Looking through this file, they see no action or method calls taking place, so they look elsewhere, in particular at `adminverify_jsp.java`.
4. They then decide to remove `adminlogin_jsp.java` from evidence after deciding it is irrelevant.
5. A range of code from `adminverify_jsp.java` is attached and E2 suggests that *Pollinator* gives the full package name path to evidence when it is being attached, to avoid ambiguity.
6. They then decide to add `adminlogin_jsp.java` back into evidence, with a text evidence explanation stating that the file is used, but only to print out the web page once logged in.
7. Evidence is marked verified, and the goal is marked as completed.

Task 5

What does the function `loadAllPosts()` *do? (post-conditions / effects of its execution.) Please also examine the effects of the* `findPost()` *method that is called.*[4]

---

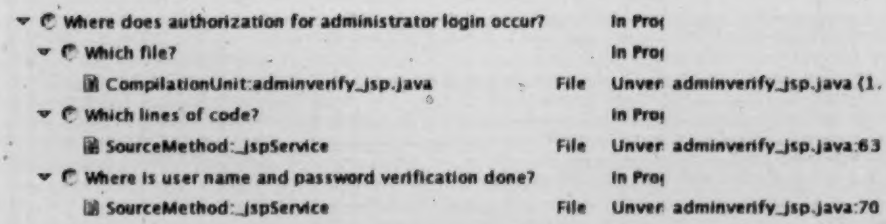
[4] The pilot study wording did not specify the inclusion of `findPost()`.



Where does authorization for administrator login occur? Complet
SourceMethod:_jspService File EVc Verified adminverify_jsp.java:65 (1.1)
CompilationUnit:adminlogin_jsp.java File EVc Verified adminlogin_jsp.java (1.1)

Name: TEXT – login procedure
Type: Text
Status: Verified
Created: Jun 15, 2007 3:29:13 PM
Last Modified: Jun 15, 2007 3:29:23 PM
Text Title: login procedure
Text Contents: Based on the attached evidence, the actual admin verification happend inside adminverify_jsp.java file and the adminlogin_jsp.java is only doing the printing job of the result of the login.
Status:

Figure C.15: Experimental study task 4 result



Where does authorization for administrator login occur? In Pro
  Which file? In Pro
    CompilationUnit:adminverify_jsp.java File Unver adminverify_jsp.java (1.
  Which lines of code? In Pro
    SourceMethod:_jspService File Unver adminverify_jsp.java:63
  Where is user name and password verification done? In Pro
    SourceMethod:_jspService File Unver adminverify_jsp.java:70

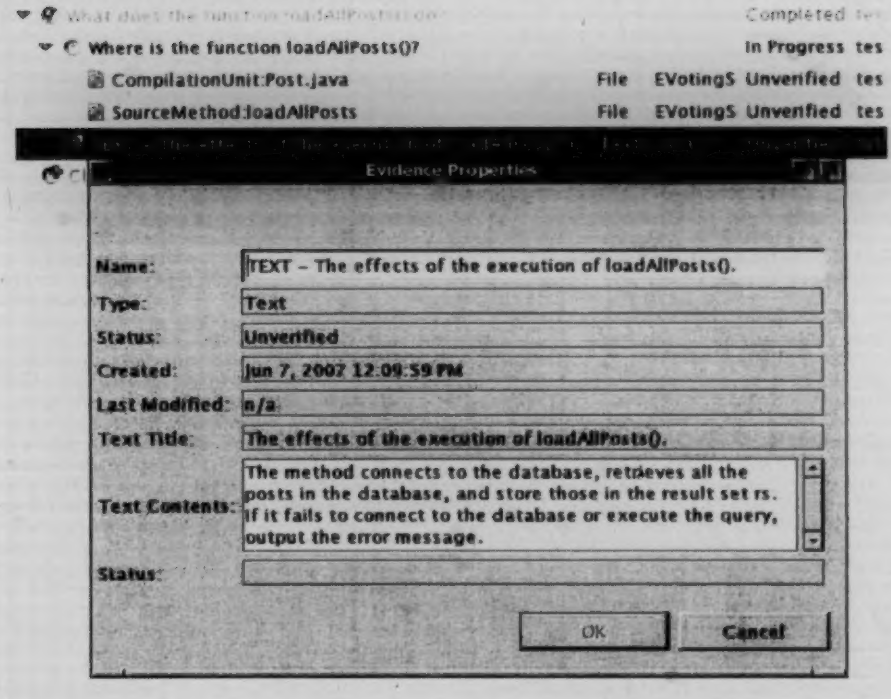
Figure C.16: Reference answer for task 4

Pilot Study

1. Participants create a root goal "What does loadAllPosts() do?".
2. They then immediately create subgoal "Where is loadAllPosts()?".
3. Now they navigate to `Post.java` and look through the member methods.
4. `loadAllPosts()` is found and dragged to the subgoal.
5. `Post.java` is also then dragged to the subgoal for the location.
6. They then browse through `loadAllPosts()` and examine it a bit, in particular making note of the use of a `ResultSet` called `rs`.
7. Text evidence is attached to the main goal as "the effects of [...] loadAllPosts()".
8. Now they go back and look at the code again.
9. They decide to alter/add to the text evidence, but this is not yet implemented in *Pollinator*, so they instead workaround by copy and pasting the original text evidence contents into a new text evidence and add the additional information.
10. Finally, they mark the goal as completed.

Control Study

1. Participants perform a file search for the string `loadAllPosts`, after an erroneously executed Java search using "Type" instead of "Method". The method `loadAllPosts()` is located in `Post.java`.
2. The pair read through the message, apparently forming a mental understanding.
3. They note a query named `findAllPosts` is referenced and go to `SQLQueries.properties` to see the raw SQL query.
4. Upon navigating back to `Post.java`, C2 has to reorient himself and is pointed by C1 to go back to `loadAllPosts()` as he had accidentally gone to `findPost()`.
5. C2 then reads out his understanding of the `loadAllPosts()` method while C1 writes it down with some of his own understanding repeated out loud and also written down.



What does the function loadAllPosts() do? Completed tes

Where is the function loadAllPosts()? In Progress tes

CompilationUnit:Post.java File EVotingS Unverified tes

SourceMethod:loadAllPosts File EVotingS Unverified tes

Evidence Properties

Name: TEXT – The effects of the execution of loadAllPosts().

Type: Text

Status: Unverified

Created: Jun 7, 2007 12:09:59 PM

Last Modified: n/a

Text Title: The effects of the execution of loadAllPosts().

Text Contents: The method connects to the database, retrieves all the posts in the database, and store those in the result set rs. If it fails to connect to the database or execute the query, output the error message.

Status:

Figure C.17: Pilot study task 5 result

6. Now, the pair navigate to `findPost()` for the second part of the task and there is discussion here again as C2 reads out the understanding and C1 writes it down, regarding the query and hash map construction.

7. C2 checks to see if there might be anything else to the method, but find nothing.

loadAllPost → gets all the post names from the DB and sets a flag to true.

findPost →
Queries the Post objects and constructs the `posts` hashmap.

Figure C.18: Control study answer to Task 5

Experimental Study

1. The pair create a goal, using a template, as "What does function loadAllPosts() do?".

2. They create a second root goal, using a template, as "What does function findPost() do?". So the participants see this task as two separate goals, initially.

3. They open `Post.java` right away, based on their prior knowledge of code naming convention and where they think `loadAllPosts()` might be located.

4. They find the `loadAllPosts()` method and attach it as evidence, including a static boolean field, to the first goal.

5. Now, they examine the guts of the method and relevant fields of `Post` trying to figure out the postconditions and effects of method execution. They do not appear to find any tangible effects, so they look at `findPost()` instead, which is called from `loadAllPosts()`.

6. While going through the code, they discover the need to find out what the difference is between the SQL query executed in `loadAllPosts()` and that in `findPost()` as they initially thought the query was the same for both.

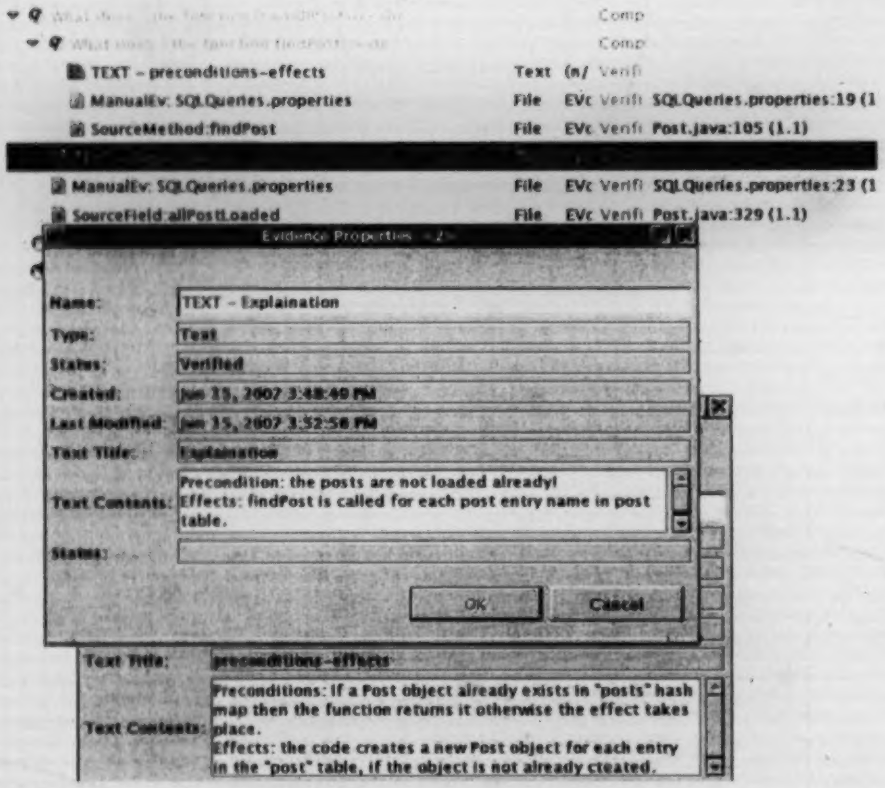
7. It is decided that the second root goal for `findPost()` can instead better be placed as a subgoal under the first root goal.

8. The `findPost()` method is attached to the subgoal along with the SQL query `findPQuery` found in `SQLQueries.properties`.

9. The SQL query `findAllPosts` is attached to the first goal.

10. Text evidence is now attached to explain `loadAllPosts()` on the root goal and separate text evidence is created for the `findPost()` subgoal to answer the specific postconditions and effects found there.

11. Evidence is marked as verified and goals marked as complete.

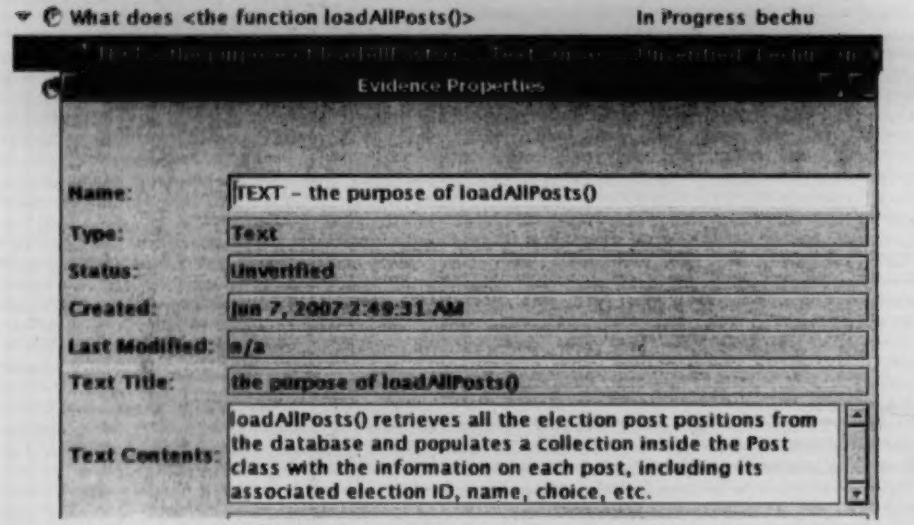


TEXT – preconditions–effects Text (n/ Verifi
ManualEv: SQLQueries.properties File EVc Verifi SQLQueries.properties:19 (1
SourceMethod:findPost File EVc Verifi Post.java:105 (1.1)
ManualEv: SQLQueries.properties File EVc Verifi SQLQueries.properties:23 (1
SourceField:allPostLoaded File EVc Verifi Post.java:329 (1.1)

Name: TEXT – Explaination
Type: Text
Status: Verified
Created: Jun 15, 2007 3:48:40 PM
Last Modified: Jun 15, 2007 3:52:56 PM
Text Title: Explaination
Text Contents: Precondition: the posts are not loaded already!
Effects: findPost is called for each post entry name in post table.
Status:

Text Title: preconditions-effects
Text Contents: Preconditions: If a Post object already exists in "posts" hash map then the function returns it otherwise the effect takes place.
Effects: the code creates a new Post object for each entry in the "post" table, if the object is not already cteated.

Figure C.19: Experimental study task 5 result



What does <the function loadAllPosts()> In Progress bechu

Name: TEXT – the purpose of loadAllPosts()
Type: Text
Status: Unverified
Created: Jun 7, 2007 2:49:31 AM
Last Modified: n/a
Text Title: the purpose of loadAllPosts()
Text Contents: loadAllPosts() retrieves all the election post positions from the database and populates a collection inside the Post class with the information on each post, including its associated election ID, name, choice, etc.

Figure C.20: Reference answer for task 5

Appendix D

User Study Participant Feedback

This appendix contains the detailed questionnaire responses given by participants of the *Pollinator* user study described in Chapter 7.

Here we group together the responses for the experimental and pilot pairs because the questionnaire for those groups was identical. The responses to the questions are reproduced here verbatim, without any alteration to their content.

Experimental and Pilot

Question 1

How difficult did you find the program comprehension task(s)? Scale: very easy, easy, average, difficult, very difficult.

P1: average

P2: easy

E1: easy

E2: easy

Question 2

Did you find that the Pollinator tool helped in the completion of the programming task(s)? Why or why not?

P1: Yes, creating goals and sub-goals make things more organized and easier to proceed.

P2: Yes, by adding goals and evidence, the user is able to attack the understanding task in a divide-and-conquer fashion, which makes the task more organizable.

E1: Yes, I do. the reason should be more evident in very large projects with multiple programmers/maintainers, where features are developed by several people. In such cases, it happens more than often that a feature needs to be changed, etc.; therefore, a tool like *Pollinator*, if used consistently during development, would be an invaluable tool.

E2: *Pollinator* helps me with setting up the goal. I think it helps me within the future if I am coming back to this project again.

Question 3

Did Pollinator help coordinate communication between you and your partner? Why or why not?

P1: Yes, because we are able to share a visual interface and knows what are the projects/goals available.

P2: Yes. The design of goals and evidences provides the users a uniform platform for discussion of the issues in the task.

E1: Yes, it did. *Pollinator* helped us come to one common structure for understanding features.

E2: Yes it helps, with these goals and sub-goals we clearly know what we are talking about.

Question 4

How usable (ease of use, clarity of action/consequence pairing, UI) is Pollinator? Scale: very bad, bad, average, good, very good

P1: very good

P2: good

E1: good

E2: good

Question 5

What did you like most about Pollinator? Why?

P1: It really helps program comprehension. The interface is organized and easy to use. Click and drop and highlight evidence is very useful.

P2: The easy-to-use drag functionalities.

E1: I liked the way it allows project developers add pieces of code as evidence for some feature (goal). That could be extremely useful in big projects.

E2: To divide the program comprehension into goals and record all answers so that in the future these answers can be quickly reloaded into one's mind.

Question 6

What did you least like about Pollinator? Why?

P1: Can't open source code from the goal explorer view.

P2: The arrangement of all the perspectives and the difficulty in switching between them.

E1: Well, the tool seemed to be very useful and easy to use, but sometimes did some wrong doings which is normal at this stage of development. And also it was minor and did not hurt the purpose of the research and tool.

E2: The editing. We have to set the goals as "verified" one by one.

Question 7

Would you use Pollinator for your own work and/or research?

P1: Yes.

P2: Yes, if it is open-source.

E1: I would use it for a big multi-developer project.

E2: Definitely. I was using a public parser for my research and every time when I went back to do the modification I have to go through the program comprehension process again. *Pollinator* can help a lot for that purpose.

Question 8

Do you have any other feedback or suggestions?

P2: When browsing the goals from the goal creating window, I hope all the goals are displayed in a leveled form, not just in alphabetical order.

E1: I think it should be easy and worthwhile to make *Pollinator* language independent; or just support other languages. I think if *Pollinator*is made a little bit formal, then formal reasoning of some feature would be possible. By formal, I mean more constrained evidences.

E2: Feedback: 1. Is it possible for us to attach some more complex explanations such as pictures or UML diagrams? So far to me, *Pollinator*can only attach text file as explanation. 2. Improve the editing (see Q6).

Control

Question 1

How difficult did you find the program comprehension task(s)? Scale: very easy, easy, average, difficult, very difficult

C1: c) average

C2: c) average

Question 2

Do you think the method of documenting program comprehension you used in this study is reliable and useful? Why or why not?

C1: The method used may not entirely be reliable as the underlying business logic was not well understood.

C2: Yes. They are useful to understand how the system works, how data is processed.

Question 3

How did this method of documenting program comprehension affect the coordination of communication between you and your partner, if at all?

C1: There was good cooperation and synergy between the teammates.

C2: I have to wait for my partners to write [down the answer].

Question 4

Do you think a specialized tool could help in the completion of the program comprehension task(s)? Why or why not?

C1: Yes. A specialized tool would contain the logic behind the design of the system and [...] has a better chance of comprehending the program successfully.

C2: A better tool support for documentation would be nice.

Question 5

Could a tool help coordinate communication between you and your partner? If so, how?

C1: Yes. It would help in orienting each of the teammates' idea of comprehension and rectify any disagreement in opinions.

C2: If two people are on the same computer, it's not a big deal. If they are in different location, a communication tool such as a video conference type tool is essential.

Question 6

Do you have any additional feedback or general comments?

C1: [No answer.]

C2: It would be nice if we have some document about the system printed out at hand.